Technická univerzita v Liberci

Fakulta mechatroniky a mezioborových inženýrských studií

Katedra softwarového inženýrství

Diplomová práce

Vybrané aplikace teorie grafů

Elected applications of the graph theory

Liberec 2004 Jan Kříž

Abstrakt

Cílem diplomové práce bylo seznámit se se základními algoritmy používanými v teorii grafů a v síťové analýze, vytvořit aplikaci, která umožní jejich vizualizaci a získané výsledky porovnat s jiným softwarem.

Byly vytvořeny jednotlivé aplikace zabývající se danou problematikou a začleněny do komplexního programu, který umožňuje výběr konkrétní aplikace a práci s ní. Vše bylo vyvíjeno pomocí programovacího jazyka Pascal ve vývojovém prostředí Borland Delphi 7.0. Tato aplikace rozšíří používané pomůcky na katedře softwarového inženýrství Technické univerzity v Liberci při výuce diskrétní matematiky a operační analýzy.

Závěry diplomové práce ukazují, že pro úlohy hledání minimálních koster grafů je nejvýhodnější použít Kruskalův algoritmus. Aplikace z oblasti síťové analýzy umožňují řešit úlohy týkající se optimalizace časových nákladů projektů metodami CPM a PERT. Dále ze srovnávacích měření provedených na vyvinuté aplikaci a na nalezeném softwaru plyne, že ač je program náročnější na paměť počítače, je v několika ohledech rychlejší. S ohledem na požadavek učební pomůcky je vytvořeno uživatelsky přátelské rozhraní.

Abstract

The aim of Diploma thesis was to acquaint with basic algorithms used in graph theory and in the net analyses, create an application that enables their visualization, and acquired results compare with another software.

Individual applications focused on given problems were developed and they are incorporated to the complex program, which allows selection of concrete particular application and work with it. Everything was developed by the help of programming language Pascal in the development environment Borland Delphi 7.0. This application will extend actually used tools for education of discreet mathematics and operation analyses at faculty of software engineering of Technical University in Liberec.

Conclusions of the diploma thesis shows, that for tasks of minimal spanning trees of graphs is best to use Kruskal's algorithm. The application from the area of net analyses enables problem solution relevant to optimization time loads of project methods CPM and PERT. Further from comparison measurement performed on developed application and on retrieval software passes, that though program is more exacting on a memory of computer, it is in some regards quicker.

The user-friendly interface is developed with a respect on requirement the teaching utility.

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé DP a prohlašuji, že **s o u h l a s í m** s případným využitím mé diplomové práce (prodej, zapůjčení apod.)

Jsem si vědom toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na náhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné velikosti).

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury na základě konzultací s vedoucím diplomové práce a konzultantem.

V Liberci dne 21.5.2004 …….

Jan Kříž

Poděkování

Rád bych poděkoval vedoucímu diplomové práce Doc.Mgr.Ing. Václavu Zádovi, CSc. za cenné rady při konzultacích, za pomoc při řešení, celkovou trpělivost a materiální i morální podporu během tvorby této práce.

Obsah

1 Úvod

Teorie grafů je praktická disciplína, která patří do relativně mladého odvětví matematiky. Její kořeny nacházíme již v 18. a 19. století, ale teprve v roce 1936 vyšla první kniha, která byla celá věnovaná teorii grafů. Mohutný rozvoj této matematické disciplíny nastal až ve druhé polovině 20. století, kdy konečně výsledky teorie grafů našly své uplatnění v mnoha praktických úlohách a úkolech co se oblasti samotné matematiky, ale také chemie, elektrotechniky, fyziky a lingvistiky týče.

Historie teorie grafů není dlouhá. Za prvotní práci je považován spis Leonharda Eulera z roku 1736 ve kterém Euler vyřešil tzv. „problém mostů města Königsbergu“. Centrum tohoto města leželo na dvou ostrovech, které s oběma břehy spojovalo sedm mostů. Šlo o nalezení cesty, která by spojovala všechny části města, začínala a končila ve stejné časti, a při které by každý most byl využit právě jednou.

Další zdroje teorie grafů nalezneme až v polovině 19. století vytvořené Gustavem Kirchhoffem, který rozpracoval některé otázky týkající se teorie stromů v souvislosti s řešením soustav lineárních algebraických rovnic, které získal při výpočtech neznámých proudů v elektrických sítích.

Nejznámější úlohou, kterou se od poloviny minulého století zabývali matematici, ale i nematematici, byl „problém čtyř barev“. Jde o ne zrovna lehký důkaz faktu, že k obarvení libovolné mapy (na kulové ploše či v rovině) je zapotřebí pouze čtyř barev. Problém byl vyřešen až v roce 1976 s využitím počítačů a od té doby podnítil k vědeckému výzkumu teorie grafů řadu matematiků i praktiků.

První práce s grafovou tématikou, která vznikla v tehdejším Československu, byla práce brněnského matematika Otakara Borůvky. Ten se zamýšlel nad problémem elektrifikace vesnic a měst jižní a západní Moravy. Měl navrhnout jejich spojení tak, aby řešení bylo tak úsporné, jak je to možné. Otakar Borůvka nejen správně formuloval tento problém, ale také ho i vyřešil a tím vytvořil první algoritmus pro nalezení minimální kostry ohodnoceného grafu. V té době neexistovalo vhodné názvosloví této oblasti matematiky a tak důkazy správnosti byly dost komplikované.

Dalším matematikem, který se zabýval tímto oborem byl Vojtěch Jarník. Třetí řešení problému, který byl odlišný od obou předchozích objevil Joseph B. Kruskal v roce 1956.

V současnosti se pozornost věnuje rychlejším a sofistikovanějším algoritmům nejen pro hledání minimální kostry grafu jak pro obyčejné grafy, tak i pro speciální třídy grafů.

2 Základní pojmy z teorie grafů

Pro seznámení se a definování některých pojmů v následujících kapitolách jsme použili literaturu [1], [2], [3] a [7].

V celé této práci se budeme setkávat s objekty, které se v grafové terminologii nazývají *grafy*. Každý graf se skládá z množiny prvků, které budeme nazývat *uzly* nebo též *vrcholy*. Spojnice některých dvou vrcholů se nazývá *hrana*, která může být *orientovaná* i *neorientovaná*.

Pokud v grafu *G* existuje hrana, která spojuje vrchol se sebou samým, nazýváme ji *smyčkou*. Neorientované grafy obsahují všechny hrany neorientované, orientované grafy mají všechny hrany orientované. Dále existují speciální grafy, které obsahují jak hrany neorientované, tak i hrany orientované. Tyto grafy nazýváme *smíšené*. Tomuto typu grafů se v celé šíři práce nevěnujeme.

2.1 Orientovaný graf

Orientovaný graf je trojice G = (*V*, *E*, *ε*) tvořená konečnou neprázdnou množinou V, jejíž prvky nazýváme *uzly*, konečnou množinou *E*, která obsahuje prvky nazývané *orientované hrany* a zobrazením ε: $E \rightarrow V^2$, kterou nazýváme *vztahem incidence*. Takovéto zobrazení přiřazuje každé hraně $e \in E$ uspořádanou dvojici vrcholů (*x*, *y*). První z nich nazýváme *počátečním vrcholem hrany* a druhý nazýváme *koncovým vrcholem hrany*.

O takovéto orientované hraně říkáme, že vede z vrcholu *x* do *y* a je incidentní s vrcholy *x*, *y*. Jedná-li se o grafy, v nichž se vyskytují smyčky, pak *x* = *y*. Vrchol, který není incidentní s žádnou hranou, nazýváme *izolovaným vrcholem*. Je také zároveň možné, aby pro různé hrany e_1, e_2 platilo $\varepsilon(e_1) = \varepsilon(e_2)$. O takových hranách říkáme, že jsou *rovnoběžné* nebo též *násobné.*

2.1.1 Symetrizace a orientace grafu

Sy*metrizace* je činnost při které z orientovaného grafu vznikne neorientovaný graf. V grafu jednoduše umažeme všechny šipky.

Máme-li zase naopak nějaký graf neorientovaný, graf orientovaný z něj můžeme vytvořit dvěma způsoby:

- V grafu libovolně zvolíme orientaci hran (přimalujeme šipky). Tímto způsobem

získáme *orientaci grafu G*.

- V grafu každou neorientovanou hranu kromě smyček nahradíme dvěma opačně orientovanými hranami. Co se týče smyček, stačí pouze přimalovat šipku. Tímto způsobem získáme symetrickou *orientaci grafu G*.

2.2 Neorientované grafy

V některých i praktických příkladech se vyskytují grafy, které obsahují hrany, jejichž orientace je nepodstatná. Takovéto situace vznikají tak, že jednoduše zapomeneme na pořadí vrcholů v uspořádaných dvojicích. *Neorientovaný* graf je trojice G = (*V*, *E*, *ε*) tvořená opět konečnou neprázdnou množinou *V*, jejíž prvky nazýváme *uzly*, konečnou množinou *E*, která obsahuje prvky nazývané *neorientované hrany* a zobrazením *ε*, které nazýváme *vztahem incidence*. Toto zobrazení každé hraně $e \in E$ přiřazuje jedno- nebo dvouprvkovou množinu vrcholů.

Říkáme, že hrana e je incidentní s těmito vrcholy nebo také jednoduše, že hrana *e* *spojuje* tyto vrcholy. Je-li hrana *e* incidentní pouze s jedním vrcholem, nazýváme ji *smyčkou*.

2.3 Ohodnocené grafy

Ve většině aplikací grafy jako takové nepostačují k popisu dané situace, kterou vyjadřují. Proto se často hranám nebo vrcholům přiřazují nějaké obvykle číselné hodnoty, které znamenají např. doby trvání, náklady činností apod. Grafy, které jsou opatřeny takovými hodnotami, nazýváme *ohodnocené grafy* nebo také *sítě*. Rozeznáváme:

$f : V \to R$ uzlově ohodnocený graf

$f : H \to R$ hranově ohodnocený graf

Pro praktické příklady se v drtivé většině používají hranově ohodnocené grafy. Pro úplnou představu uveďme jeden praktický příklad:

Představme si dopravní např. silniční síť. Tuto síť lze popsat grafem *G*, jehož každý uzel bude odpovídat městu a hrany zase jednotlivým cestám. Budeme-li chtít řešit praktické otázky silniční dopravy pomocí našeho takto vzniklého grafu, je přirozené i logické každé hraně našeho grafu *G* přiřadit libovolně zvolené reálné číslo, které bude představovat např. délky mezi jednotlivými městy (uzly).

2.4 Výčet některých typů grafů

Úplný graf – graf, ve kterém jeho každé dva vrcholy jsou spojeny hranou. Pokud graf obsahuje *n* vrcholů, pak počet hran je roven *n * (n - 1) / 2*.

Prostý graf – graf, v němž násobnost každé hrany dosahuje hodnoty nejvýše jedna, čili nemá vícenásobné hrany.

Rovinný graf – graf, jehož uzly jsou body roviny a hrany lze reprezentovat spojitými orientovanými neprotínajícími se křivkami ležícími v rovině.

Stupeň vrcholu – číslo rovnající se počtu hran incidentních s daným vrcholem. Posloupnost stupňů jednotlivých vrcholů pak nazýváme *skóre grafu*.

Sled – sled délky k v grafu G je posloupnost navazujících vrcholů a hran $(v_0, e_1, v_1, ..., e_k, v_k)$ taková, že $e_i = \{v_{i-1}, v_i\}$ pro $i = 1, ..., k$.

Tah - tah délky k v grafu G je sled délky k v grafu G takový, že platí $e_i \neq e_j$ pro každé $i \neq j$, $i, j = 1, \ldots, k$. Jedná se o sled, ve kterém se neopakují žádné hrany.

Cesta - cesta délky k v grafu G je sled délky k v grafu G takový, že platí $v_i \neq v_j$ pro každé $i \neq j$, $i, j = 0, \ldots, k$. Jedná se o sled, ve kterém se neopakují žádné vrcholy.

Podgraf – graf *G'* vznikne z grafu *G* prostým vynecháním některých hran nebo uzlů. Touto operací musí vzniknout opět graf.

Komponenta grafu – souvislý podgraf grafu *G*, který není obsažen v žádném větším souvislém podgrafu grafu *G* (tj. komponenta grafu *G* je každý maximální souvislý podgraf grafu *G*).

Souvislý graf – graf, pro který platí, že mezi každými dvěma jeho vrcholy existuje cesta. Takový graf se skládá z jedné komponenty.

Faktor grafu G' vznikne z grafu *G* prostým vynecháním některých hran. Potom platí, že *V(G) = V(G')*.

Strom - graf, který je souvislý a navíc neobsahuje smyčku.

Les - graf, který neobsahuje kružnici.

Kostra v neorientovaném ohodnoceném souvislém grafu *G* je faktor grafu, který je stromem.

Uzavřený sled - resp. uzavřený tah délky *k* v grafu *G* je sled (resp. tah) délky *k* v grafu *G* takový, že vrchol, z kterého vycházíme (první vrchol sledu / tahu), je roven vrcholu, ve kterém končíme (poslední vrchol sledu / tahu).

Uzavřená cesta je uzavřený sled, v němž se neopakují vrcholy ani hrany. Pro tyto pojmy se používají speciální názvy:

- *kružnice* je neorientovaná uzavřená cesta
- *cyklus* je orientovaná uzavřená cesta

Most – hranu *e* grafu *G* nazýváme most, jestliže graf *G* – *e* má více komponent než graf *G*. Jedná se tedy o hranu, kterou když vyjmeme, dojde k navýšení počtu komponent (graf se rozpadne na více částí). Také je to hrana, která neleží na žádné kružnici.

2.5 Reprezentace grafů maticemi

Jedná se o několik odlišných přístupů, jak lze přistupovat ke struktuře grafu.

2.5.1 Matice sousednosti

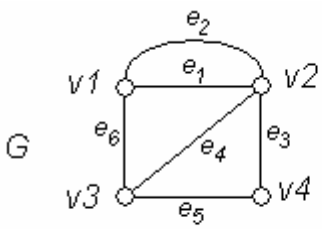
Nechť G je orientovaný graf. Zvolíme – li pevně pořadí jeho vrcholů $v_1.....v_n$, můžeme grafu *G* přiřadit *matici sousednosti* M_G^+ řádu *n* s předpisem $m_{ij}^+ = m^+(v_i, v_j)$. Pro neorientované grafy definujeme matici sousednosti předpisem $m_{ij} = m(v_i, v_j)$.

2.5.2 Matice incidence

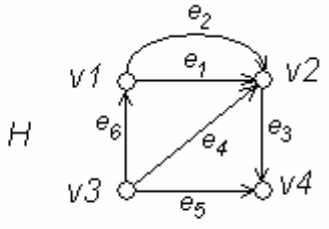
Nechť *G* je orientovaný graf bez smyček. Zvolíme – li pevně nejen pořadí vrcholů $v_1.....v_n$, ale i pořadí hran $e_1.....e_n$, můžeme grafu *G* přiřadit *matici incidence* B_G předpisem

$b_{i,j}$ = 1, jestliže v_i je počátečním vrcholem hrany e_j
$b_{i,j}$ = -1, jestliže v_i je koncovým vrcholem hrany e_j
$b_{i,j}$ = 0, v ostatních případech

Uvedeme příklad:



Obrázek 1.



Obrázek 2.

Matice sousednosti grafů G, H budou tedy vypadat následujícím způsobem:

$$M_H = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \qquad M_G^+ = \begin{pmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

a matice incidence orientovaného grafu:

$$M_H = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & -1 \\ -1 & -1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & -1 & 0 & -1 & 0 \end{pmatrix}$$

Jelikož jsme tuto matici ve své práci dále nevyužili, nebudeme se k ní již vracet.

2.6 Různé možnosti zadávání grafů

Nejčastěji se grafy zadávají vizuálně, nakreslené pomocí rovinného kreslení. Pochopitelně existují prostředky (např. aplikace Excel), které umožní tuto 2D grafiku převádět na 3D a vyrobit tak prostorový vzhled grafu. Otázkou ovšem je, zda nad takovým zadáním realizovaným v počítači dovedeme provádět běžně úlohy nad grafy. Zadávání může být i takové, že není zatím plně vhodné pro počítačové zpracování. Obecně je potřeba rozlišovat vlastní zadání grafu a jeho prezentaci datovými strukturami pro zpracování v počítači.

2.7 Zadání grafů vhodné pro počítačové zpracování

Dále uvedená *zadání grafů* se dají použít k počítačovému zpracování. Některá z nich mají své přednosti, jiná zase nedostatky.

2.7.1 Seznam hran a svázaných uzlů

Graf je zadán ve tvaru seznamu $h_1{:}x_1,y_1;\ h_2{:}x_2,y_2;\ h_3{:}x_3,y_3;\ \ldots\ h_n{:}\ x_n,y_n$; kde h_i značí i - tou hranu. Pořadí uzlů hrany definuje její orientaci. Je-li seznam velký, potom je v něm obtížné hledání. Vhodné je tento seznam nějak uspořádat, např. podle ucelených řetězců navazujících hran. Obráceně je seznam vhodný pro řídké grafy (grafy s málo hranami), protože je úsporný.

2.7.2 Matice sousednosti

Je matematicky elegantní, ovšem pro rozsáhlé grafy je v matici mnoho nul. To zbytečně prodlužuje práci s grafem. Na druhé straně většina programovacích jazyků dovede velmi jednoduše matice deklarovat jako pole a pracovat s nimi (operace).

2.7.3 Matice incidence

Má podobnou charakteristiku jako matice sousednosti.

3 Problém minimální kostry grafu

V této části práce jsme čerpali z literatury [1], [2], [3] a [7].

Představme si, že máme dán souvislý graf *G*, jehož hrany jsou ohodnoceny reálnými čísly, které budeme nazývat cenami. *Kostru grafu*, která má nejmenší součet ohodnocení hran mezi všemi kostrami, nazýváme nejlevnější nebo též minimální kostrou.

V praxi se většinou setkáváme s ohodnocením hran nezápornými čísly. My pro jednoduchost zvolíme ještě větší omezení a to, že se zaměříme na ohodnocování hran pouze celými kladnými čísly.

3.1 Hladový algoritmus - Kruskalův

Následující postup hledání nejlevnější kostry popsal J.B. Kruskal v roce 1956.

Hrany grafu uspořádáme podle jejich cen do neklesající posloupnosti. Pak je v tomto pořadí probíráme a do postupně vytvářeného grafu *L* přidáváme ty z nich, které v grafu *L* nezpůsobí vznik kružnice.

Kruskalův algoritmus bývá velmi často používán pro názorné řešení příkladů hledání minimální kostry grafu, protože je systematický a přehledný.

Algoritmus:

Použijeme pole uzlů U_Z, kde si označíme vrcholy začleněné do lesu a pole hran H_R, kde si označíme hrany, začleněné do nejlevnější kostry. Do proměnné *k* uložíme počet hran v lese.

1. Inicializace : $U_{Zl} = 0$; $H_{Rl} = 0$ pro $\forall i$ a $k = 0$;
2. Ohodnocené hrany setřídíme do neklesající posloupnosti.
3. Je – li $k = n-1$, potom bod 5. Jinak vezmeme nejlevnější hranu h_i takovou, že $H_{Rl} = 0$.
4. Testujeme, zda můžeme hranu $h_i = (u_X, u_Y)$ přidat do lesu, aniž by vznikla kružnice.
   Je – li $(U_{ZX} = 1 \wedge U_{ZY} = 1)$ testujeme (viz poznámka), zda přidáním hrany h_i nevznikne v lesu L kružnice. Vznikne – li kružnice, pak hranu h_i nemůžeme přidat do kostry a změníme $H_{Rl} = 2$. Jinak $\{ U_{ZX} = 1; U_{ZY} = 1; k = k+1; H_{Rl} = 1 \}$ a pokračujeme bodem 3.
5. Konec.

Poznámka:

V poli H_R máme uloženy informace o hranách, které jsme ještě nepoužili (0), které byly přidány do kostry (1) a které by způsobily vznik kružnice (2). Z tohoto příkladu vyplývá, že máme – li v seznamu více hran, které mají shodné minimální ohodnocení, existuje tudíž více minimálních koster, důkaz viz. [3].

Odvození časové náročnosti algoritmu:

- 1. ř: O (1)
- 3. ř O (m) operací lze implementovat tak, že každá vezme O (log m), celkem tedy O(m.log m), přesně O(n). (O(1) + O(log m)).
- Celkem tedy:

  $$O(n) + O(m.\log m) = O(m.\log m)$$
- Cyklus 4. ř se provádí |m| - krát.

$$O(\log m) \Rightarrow O(m.\log m) = O(m.\log n)$$

3.2 Jarník – Primův algoritmus

Stejný postup řešení problému minimální kostry odvodil v roce 1957 R.C. Prim.

Předpokládáme, že na počátku algoritmu není žádná hrana obarvená a graf *G* je diskrétní. Pro každou hranu *e* nakresleného grafu G máme dané celé kladné číslo *w(e)*. Zvolíme libovolný počáteční uzel *v* z množiny všech uzlů grafu *G* a obarvíme ho zeleně. Tímto postupem získáme základní počáteční zelený strom. Pro tento strom hledáme v každém kroku *k* (*k = 1, 2,….,n-1*) množinu všech hran incidentních s tímto stromem. Z takto vzniklých množin hran vybereme ovšem jen tu, která má minimální ohodnocení. Tato hrana má jeden svůj uzel již zahrnutý v zeleném stromu a druhý nikoliv. Tuto hranu přidáme k aktuálnímu stromu. Jestliže existuje více hran, které mají shodné minimální ohodnocení, pak vybereme libovolnou z nich.

algoritmus:

1. [Inicializace kostry *K*]
   Nechť *v* je libovolný vrchol grafu *G*
   V(K) = {*v*}, *H(K)* = 0
2. [Aktualizace kostry]
   Nechť *h* je hrana s minimálním ohodnocením z hran spojujících vrchol $u \in V(K)$ s vrcholem $v \notin V(K)$, pak $V(K) = V(K) \cup \{v\}$ a $H(K) = H(K) \cup \{h\}$
3. [Test ukončení]
   Jestliže *H(K) = V(G)-1*, pak *K = (V, H(K))*, jinak návrat na krok 2.

Budeme – li chtít tento algoritmus naprogramovat do některého z programovacích jazyků a použijeme – li vhodných datových struktur, může být Jarník – Primův algoritmus implementován v čase O (n^2).

3.3 Borůvkův algoritmus

Jedná se o nejstarší algoritmus, který byl v roce 1926 popsán Otakarem Borůvkou. Pro názornost tento algoritmus opět aplikujeme na souvislý, ohodnocený graf *G*. Dále předpokládáme, že různým hranám jsou přiřazena různá čísla.

Na počátku máme všechny hrany grafu *G* neobarvené a graf je diskrétní, tzn. že jednotlivé vrcholy tvoří jednoduché, samostatné komponenty zeleného stromu. V každém kroku algoritmu zvolíme pro každý zelený strom mezi všemi hranami incidentními s tímto stromem právě tu hranu, která má nejmenší ohodnocení. Všechny zvolené hrany

obarvíme zeleně a vždy sjednotíme zelené stromy obsahující koncové uzly zeleně obarvené hrany v jeden zelený strom.

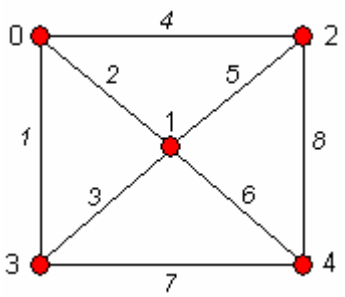
Pro lepší představu si činnost algoritmu můžeme představit tak, že graf *G* pokrýváme souborem „bublinek“. Nacházíme mezi nimi nejkratší spojení a každé takové spojení vede k tomu, že se bublinky v konečné fázi spojí v jednu.

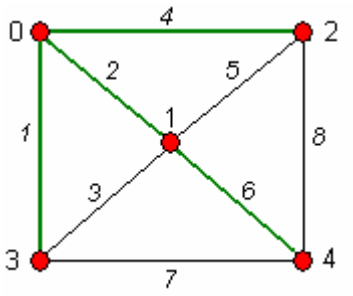
algoritmus:

1. [*Inicializace lesa L*]
   Nechť *v* je libovolný vrchol grafu *G*
   V(L) = V(G), H(L) = 0
2. [*Aktualizace lesa L*]
   Pro každou komponentu *L'* lesa *L* určit nejlevnější hranu z hran vycházejících z *L'* k jiným komponentám. Nechť *S* je množina těchto hran. Pak položme
   H(L) = H(L) $\cup$ *S* a určeme komponenty souvislosti v lese *L* = (*V*, *H(L)*).
3. [*Test ukončení*]
   Jestliže *H(L) = V(L)-1*, pak *K = (V, H(L))*, jinak návrat na krok 2.

Počet kroků, které tento algoritmus spotřebuje na nalezení minimální kostry grafu G, je menší než $\log_2$ n, protože v každém kroku se počet stromů sníží nejméně o polovinu. Použitím jednoduchých datových struktur lze algoritmus implementovat v čase O(m.log n).

Jelikož všechny algoritmy pro hledání minimální kostry grafu budou při použití na stejný graf stejné, uvedeme zde pouze jeden (Kruskalův) algoritmus. Jedná se o vizualizaci Kruskalova algoritmu.





Závěrem bychom mohli říci, že z Borůvkovy metody vychází optimální řešení za předpokladu, že žádné dvě hrany nejsou ohodnoceny stejně (Důkaz viz [1]). Dnes se ukazuje, že v Borůvkově práci komplikovaně vyjádřený algoritmus je mnohem lepší a perspektivnější, než se v minulosti předpokládalo.

Poznámka:

Všechny tyto algoritmy řeší hledání minimální kostry grafu. O tom, že vzniklá kostra bude opravdu minimální, se čtenář může přesvědčit v [1].

Jestliže považujeme váhu hrany za číslo, které udává vzdálenost mezi dvěma vrcholy, potom základní rozdíl mezi těmito třemi algoritmy může být charakterizován takto:

Kruskalův algoritmus spojuje dva nejbližší zelené stromy v jeden zelený strom. V každém kroku rozhodujeme právě o jedné hraně, zda bude do stromu přidána či nikoliv.

Jarníkův algoritmus v každém kroku pouze rozšíří zelený strom, který obsahuje počáteční vrchol následovaný nejbližším vrcholem.

Borůvkův algoritmus v každém kroku provede spojení všech zelených stromů, které jsou si navzájem nejblíže.

Efektivnost, tedy jednotlivé časové náročnosti všech tří algoritmů, uvádíme v kapitole [10.5].

4 Prohledávání grafů

V této části práce jsme čerpali z literatury [1], [2] a [3].

Základním účelem prohledávání grafů je zjišťování dostupnosti, tzn. zjistit, do kterých vrcholů grafu vedou cesty z daného výchozího vrcholu a zároveň také zjistit, kterými vrcholy a hranami procházejí.

Jsou známy tři základní algoritmy prohledávání. První z nich (značkování vrcholů) je dosti obecný. Tento algoritmus jsme neimplementovali do programu, proto ho zde nebudeme ani vysvětlovat. Další dva způsoby, prohledávání do hloubky a do šířky, lze pokládat za jeho speciální případy.

4.1 Prohledávání grafu do šířky

V grafu najde všechny vrcholy, do nichž vede cesta ze zadaného vrcholu r a cesty přitom obsahují nejmenší počet hran.

Algoritmus :

1. [*Inicializace*]

for $\forall$ *V* - {*r*} **do begin**

dosažen [v] := false; *počet-hran-od-r [v]* := ∞; *předchůdce [v]* := nil;

end;

počet-hran [r] := 0; *dosažen [r]* := true; *FRONTA* := [*r*];

while 2. [*Test ukončení*] *FRONTA* $\neq$ () **do begin**

3. [*Volba vrcholu ze začátku fronty*]

v := *zač-fronty*;

4. [Postup do šířky]

for $\forall$ $V_G^+(v)$ **do** { $V_G(v)$ ve verzi pro neorientovaný graf}

if not (*dosažen [w]*) **then begin**

dosažen [w] := true; *předchůdce [w]* := *v*;

počet-hran-od-r [w] := *počet-hran-od-r [v]*+1;

FRONTA := FRONTA + (*w*);

end;

FRONTA := FRONTA - (*v*);

end;

Časová složitost algoritmu je O(m+n), protože operace vložení vrcholu na konec fronty a odebrání vrcholu ze začátku fronty mají složitost O(1). Pro všechna provedení cyklu **while** pak O(n). Souhrnná délka seznamů následníků $V_G^+(v)$ je ohraničena počtem hran, tj. časem O(m).

4.2 Prohledávání grafu do hloubky

Hledání do hloubky si lze představit jako průzkum grafu cestovatelem, který cestuje po hranách grafu a vždy se důsledně vrací cestou, po které přišel. V grafu najde všechny vrcholy, do nichž vede cesta ze zadaného vrcholu *r*.

Algoritmus:

1. [*Inicializace*]

for $\forall$ $v \in V$ **do begin**

stav [v] := nedosažen; *předchůdce [v]* := nil;

end;

i := 0;

procedure postup_do_hloubky (r)

stav [v] := otevřen; *i* := *i* + 1; *poprvé[v] := i*;

for $\forall w \in V_G^+(v)$ **do** { $V_G^+(v)$ ve verzi pro neorientovaný graf}

if *stav[w]* = nedosažen **then begin**

předchůdce [w] := *v*;

postup_do_hloubky (w)

end;

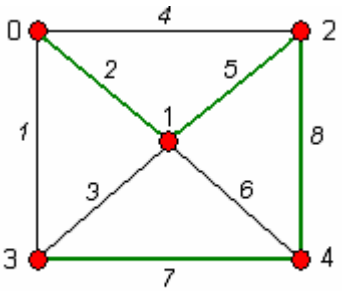
stav [v] := zavřen; *i* := *i* + 1; naposled*[v] := i*;

Časová složitost algoritmu je O(m+n). První cyklus **for** $\forall v \in V$ **do** má časovou složitost O(n). Během provádění procedury *postup_do_hloubky (v)* se cyklus **for** provádí $V_G^+(v)$*-krát.* Celkový čas cyklu **for** v proceduře *postup_do_hloubky (v)* je roven O(m).

Při hledání do šířky je podstatné, že ze seznamu *FRONTA* odebíráme vždy ten vrchol, který je v něm nejdéle. Naopak při prohledávání do hloubky ze zásobníku odebíráme vždy hranu, která je v seznamu nejkratší dobu. Hledání do hloubky je základem řady dalších velmi výkonných algoritmů, viz. kapitola 8.

Pro snadnější představu uvádíme výsledky na prohledávání grafu nejdříve do šířky a potom do hloubky, které jsme dostali na základě originálu[1]. V těchto dvou příkladech se pro jednoznačnost budeme řídit konvencí, podle níž v každém kroku, kdy je několik možností výběru uzlu, algoritmus zvolí uzel s nejmenším číslem.

Prohledávání do šířky



Prohledávání do hloubky

[1] V této podobě lze obdržet výsledky i z vytvořené aplikace.

5 Porovnávání grafů

V této části práce jsme čerpali z literatury [1], [2] a [3].

Velmi často se grafy porovnávají. Tentýž graf může být nakreslen v rovině několika způsoby. Některé z takto provedených zakreslení vizuálně vypadají odlišně, ačkoliv zakreslení zachovává podstatné vlastnosti původního grafu.

Rovnost grafů – dva grafy G a G' jsou si rovny, když $V = V'$, $E = E'$ a $\varepsilon = \varepsilon'$, tzn. že tyto dva grafy mají stejný počet uzlů, hran i vztah incidence.

Isomorfismus grafů - Grafy G, G' se nazývají vzájemně isomorfní, když existují vzájemně jednoznačná zobrazení f: $V \rightarrow V'$ a g: $E \rightarrow E'$ taková, že zachovávají vztahy incidence ε a ε'. Vztah isomorfismu značíme $G \cong G'$.

Při zjišťování isomorfismu se většinou stačí omezit na jednoduchá pozorování:

1. Isomorfní grafy musí mít stejný počet vrcholů i hran.
2. Vrcholu stupně k lze přiřadit zase jen vrchol stejného stupně k.

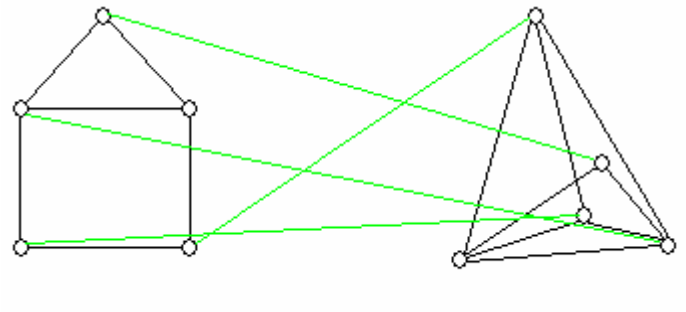
Máme - li porovnávat např. dva „malé“ grafy, není zpravidla těžké uhodnout, jestli odpovídají isomorfním grafům nebo ne, i když grafy nemusí zdaleka vypadat na pohled stejně. Nicméně úloha rozhodnout, zda dané dva grafy jsou isomorfní, je obecně obtížná a není pro ni znám žádný efektivní algoritmus[2].

Implementační část

Navržená aplikace pracuje tedy tak, že načte dva binární soubory, vykreslí je na obrazovce a zjistí o nich, zda jsou nebo nejsou isomorfní, tj. program se pokusí najít takové přiřazení mezi vrcholy grafů, že odpovídající vrcholy mají stejný počet sousedů (zřejmě by oba grafy měly mít i stejný počet vrcholů).

Na hledání jednoznačného přiřazení mezi vrcholy grafu jsme použili algoritmus prohledávání s návratem (backtracking), který pracuje takto: pokusíme se prvému vrcholu G_1 přiřadit nějaký vrchol z G_2 (musí mít stejný počet sousedů), potom druhému vrcholu G_1 nějaký z G_2 (který ovšem ještě nebyl přiřazený a má také stejný počet sousedů), atd.

[2] tj. fungující efektivně ve všech případech.



Na obrázku je vidět, jak skončilo hledání vzájemného isomorfismu dvou grafů.

6 Hledání nejkratší cesty

V této části práce jsme čerpali z literatury [1], [2], [3] a [7].

Jednou ze základních algoritmických úloh v teorii grafů je najít nejkratší cestu mezi dvěma vrcholy v daném grafu. Takovýto požadavek vzniká v mnoha praktických aplikacích, třeba při hledání nejkratšího dopravního spojení vlakem nebo po silnici apod.

Jeden z nejjednodušších a nejdůležitějších je tzv. *Dijkstrův algoritmus*.

Přepokládejme, že máme zadaný graf *G*, který má hrany ohodnocené celými kladnými čísly. Délka nějaké cesty v takto ohodnoceném grafu je rovna součtu délek jejích hran a vzdálenost *d(u,v)* dvou vrcholů *u* a *v* je pak rovna nejmenší z délek všech cest, spojujících *u* a *v.* Vstupem tohoto algoritmu je graf *G*, ohodnocení hran *w*: $E \rightarrow (0,\infty)$ a počáteční vrchol „*start*“ (s). Algoritmus pro každý vrchol $v \in V(G)$ vypočítá číslo *d(s,v)*, tj.vzdálenost z *s* do *v*. Pro každý vrchol tohoto grafu se zavede jedna číselná proměnná *d(v)*. To je číslo udávající momentální odhad hledané hodnoty *d(s,v)*. V každém kroku se pro některé vrcholy *v* momentální hodnota *d(v)* prohlásí za definitivní a pro ostatní vrcholy se *d(v)* přepočítává.

algoritmus použitý v aplikaci:

množiny hran:
1 - Obsahuje hrany, které jsou již vybrány.

2 - Hrany, z nichž právě jedna bude v příštím kroku přidána do mn. *1*.

3 - Ostatní hrany.

množiny uzlů:
A - Uzly příslušné k hranám z mn. *1*. (Pro každý uzel z mn. *A* registrujeme jeho minimální

doposud známou vzdálenost od startu *s*.)

B - Uzly příslušné k hranám z mn. *2*.

C - Ostatní uzly.

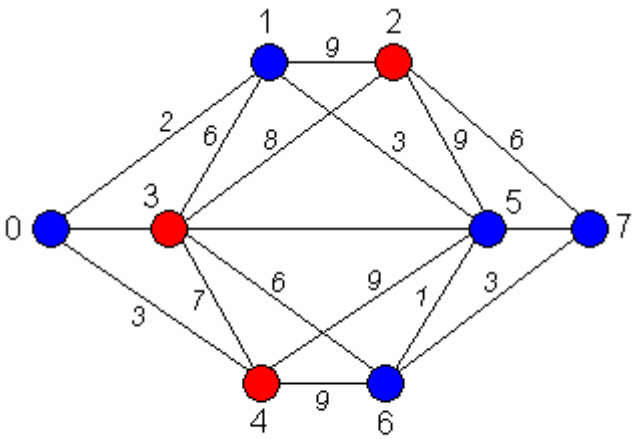
1. krok:

Start dejte do mn. *A*. Všechny hrany vycházející ze *startu* dejte do mn. *2* a koncové uzly těchto hran dejte do *B*.

- Proveď následují krok tolikrát, až v *A* bude *cíl:*

2. krok:

Každý uzel z *B* je incidentní s právě jednou hranou z *2*, jejíž počáteční uzel je v A. Tím je definovaná jistá dráha ze startu do každého uzlu v B. Uzel x_i z *B*, do kterého jde nejkratší dráha ze *startu*, odstraníme z *B* a přidáme do *A*. Odpovídající hranu z *2* odstraníme a přidáme ji do *1*. Všechny orientované hrany, které vycházejí z x_i a jejichž koncové uzly leží v *C*, odstraníme z *3* a přidáme do *2*. Koncové uzly těchto hran přidáme k *B*. Nyní uvažujeme všechny orientované hrany *h*, které vycházejí z x_i a jejichž koncové uzly leží v *B*, a zkoumáme, zda-li užitím některé hrany *h* nezkrátíme již známou dráhu ze startu do uzlů v *B*. Je-li tomu tak, pak hranu *h* odstraníme z *3* a přidáme k *2* a odpovídající hranu odstraníme z *2* a přidáme do *3*.



Nejkratší cesta z vrcholu 0 do vrcholu 7 s délkou 9.

Tento algoritmus lze použít i pro orientované grafy. Pro algoritmus platí časový odhad $O(n^2)$, protože $n + (n-1) + (n-2)+...+2+1 = \frac{n}{2}.(1+n) = O(n^2)$

7 Eulerovské grafy

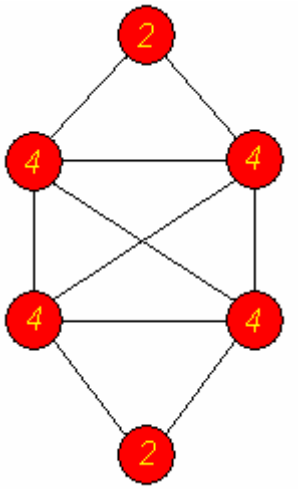
Pro seznámení se a definování některých pojmů v následujících kapitolách jsme použili literaturu [1] a [3].

7.1 Obyčejný eulerovský graf

Tyto úlohy lze stručně vysvětlit tak, že je třeba nakreslit daný graf jedním uzavřeným tahem bez zvednutí tužky z papíru. To lze matematicky zformulovat tak, že máme za úkol nalézt uzavřený sled ($v_0, e_1, v_1, ..., e_k, v_k$), v němž se každá hrana grafu G vyskytuje právě jednou a každý vrchol grafu G alespoň jednou. Takový sled budeme nazývat *uzavřeným eulerovským tahem* a příslušný graf *eulerovským grafem*.

Graf, jehož všechny vrcholy mají sudý stupeň, budeme nadále pro zjednodušení vyjadřování nazývat *sudým grafem*. Pro jakýkoliv sudý graf platí, že neobsahuje most. Pokud tedy vyjmeme libovolnou hranu takového grafu, neporušíme tak jeho souvislost. Každá hrana sudého grafu leží na nějaké kružnici.

Aby byl graf eulerovský, musí splňovat *podmínku sudosti*. Druhá vlastnost, která musí být splněna, je ta, že takový graf musí být souvislý. To znamená, že je tvořen jednou jedinou komponentou.



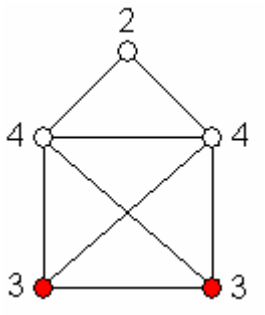
Na obrázku si můžeme prohlédnout příklad takového eulerovského grafu. Jelikož se tento graf skládá z jediné komponenty, je souvislý. Čísla v kolečku představují stupně jednotlivých vrcholů. Protože tyto čísla nabývají hodnot pouze 2 a 4, jedná se o sudý graf.

7.2 Jednotažka

Jednotažka je graf, v němž existuje tah $(v_0, e_1, v_1, ..., e_k, v_k)$ obsahující všechny hrany grafu. Jednotažka, zjednodušeně řečeno, je graf, který lze nakreslit jedním tahem (ne nutně uzavřeným).

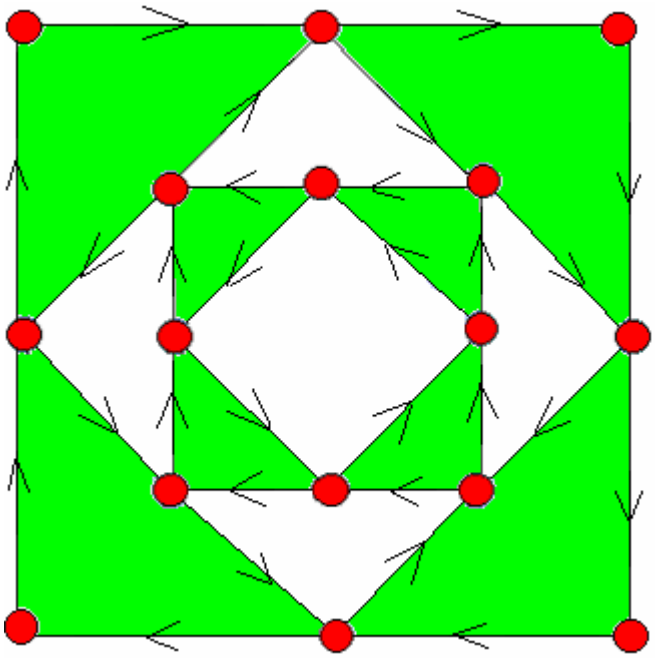
Graf je jednotažka právě tehdy, když je souvislý a zároveň je buď sudý nebo má právě dva vrcholy lichého stupně $\Rightarrow$ každý graf je jednotažka.

Klasický příklad jednotažky, která má právě dva uzly stupně lichého, je na obrázku níže.



V tomto případě se jedná o vrcholy červeně obarvené, jejichž stupně jsou rovny třem. Pokud bychom hledali tah tímto grafem, musíme vždy vycházet z jednoho z nich a končit v druhém lichém vrcholu. Tyto tahy nazýváme *otevřené eulerovské tahy.*

Orientovaný graf *G* se nazývá *vyvážený*, jestliže pro každý vrchol grafu *G* platí, že vstupní stupeň vrcholu se rovná výstupnímu stupni tohoto vrcholu.

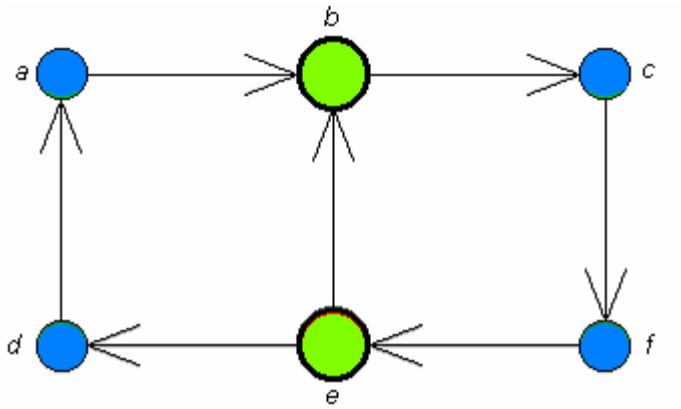


Tento obrázek zachycuje orientovaný eulerovský graf. Je vidět, že pro jakýkoliv jeho vrchol počet vstupujících a vystupujících šipek se rovná. Graf je tedy vyvážený.

7.3 Orientovaná jednotažka

Orientovaná jednotažka je graf orientovaný, mající obdobné vlastnosti jako obyčejná jednotažka. Je to souvislý graf, který má buď všechny vrcholy vyvážené, nebo existují právě dva vrcholy lichého stupně s následující vlastností:

- u prvního vrcholu je rozdíl mezi vstupním stupněm a výstupním stupněm roven jedné
- u druhého vrcholu se jedná o přesně opačnou záležitost – rozdíl výstupního a vstupního stupně tohoto vrcholu je roven jedné. V druhém případě bude tah vždy vycházet z toho lichého vrcholu, který má výstupní stupeň větší než vstupní stupeň.



Přehledně vše ukazuje obrázek, ve kterém vrcholy *b* a *e* jsou lichého stupně. Do vrcholu *b* vstupují dvě šipky a vystupuje jedna. V případě vrcholu *e* je tomu naopak (vstupuje jedna a vystupují dvě šipky). Z předchozí definice vyplývá, že při kreslení tahu musíme vždy vycházet z vrcholu *e*, ve kterém počet vystupujících šipek převyšuje počet šipek vstupujících.

Implementační část

Pokud byl vytvořen nějaký graf, máte možnost na něm otestovat algoritmus *eulerovského tahu*. Jelikož náplní této aplikace jsou eulerovské grafy, jedná se tedy o algoritmus eulerovského tahu. Vytvořená aplikace si také samozřejmě poradí s jednotažkou, orientovaným eulerovským grafem a orientovanou jednotažkou.

Pro algoritmus kreslení eulerovského tahu budeme potřebovat datovou strukturu zásobník (jedná se o datovou strukturu typu LIFO). Do zásobníku budeme postupně ukládat vrcholy grafu, kterými budeme právě procházet. Když dojdeme do vrcholu, odkud již nepovede žádná dosud nepoužitá hrana, tento vrchol ze zásobníku vyjmeme a opět se zajímáme o vrchol ležící na konci zásobníku. To vlastně znamená, že se vrátíme poslední použitou hranou zpět. Tento návrat zaznamenáme do další datové struktury, kterou označíme E (eulerovský tah). Tj. ze zásobníku vyjímané vrcholy odkládáme do E.

Při vykonávání algoritmu v aplikaci jsou vrcholy / hrany obsažené v pomocném zásobníku vykreslovány modře a vrcholy/hrany obsažené v zásobníku *E* se zobrazují červeně.

V algoritmu na konstrukci uzavřených eulerovských tahů se střídavě vykonávají dvě fáze:

- Existující tah se snažíme prodloužit a prodlužujeme ho do té doby, dokud se nestane uzavřeným.
- Uzavřený tah kontrolujeme, zda je eulerovský.

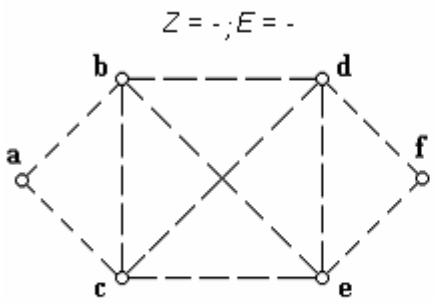
Při kontrole tedy procházíme podél tahu a v každém vrcholu *x* testujeme hranu, která dosud není obsažena v tahu. Tato hrana musí splňovat podmínku, že vrchol *x* je jejím počátečním vrcholem. Pokud nějakou takovou hranu *h* najdeme, kontrolu přerušíme, tah ve vrcholu *x* rozpojíme a začneme jej prodlužovat počáteční hranou *h*. Prodlužování skončí ve vrcholu *x*. Po spojení „staré“ a „nové“ části tahu pokračujeme v kontrole od vrcholu *x*.

Obyčejný eulerovský graf

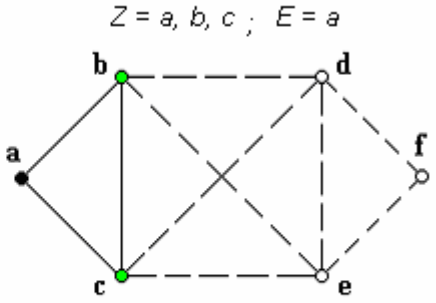
Na obrázcích níže je zachycen průběh kreslení eulerovského tahu. Hrany doposud nepoužité jsou označeny přerušovanou čarou. Hrany, které jsme prošli při přidávání vrcholů do zásobníku Z, jsou označeny plnou čarou a hrany tvořící samotný eulerovský tah E jsou zvýrazněny tučně.

Na začátku jsou obě datové struktury prázdné (obrázek 1). Zvolíme si třeba vrchol *a* a vložíme jej do zásobníku. Z vrcholu *a* vedou dvě hrany, vybereme tu do vrcholu *b* (postupujeme podle abecedy). Tento vrchol opět vložíme do zásobníku. Z vrcholu *b* se přes vrchol *c*, který též vložíme do zásobníku, dostaneme zpět do vrcholu *a* a vložíme jej

do zásobníku. Z vrcholu *a* však další hrana nevede, proto přesuneme tento vrchol do E (obrázek 2).

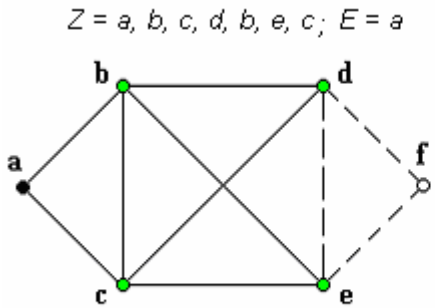


Obrázek 1.

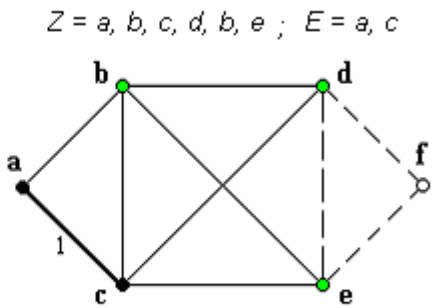


Obrázek 2.

Posledním vrcholem v zásobníku Z je v tuto chvíli vrchol *c*. Pokračujeme bez potíží do vrcholů *d, b, e* a znovu do vrcholu *c* (obrázek 3). Zjistíme, že hrany incidentní s vrcholem *c* jsou nyní již vyčerpány (všechny jsou označeny plnou čarou). Vyjmeme ho a vložíme jej do E, kde se nyní nacházejí vrcholy *a, c* (obrázek 4), které zároveň určují zařazení hrany {*a, c*} do výsledného tahu E.

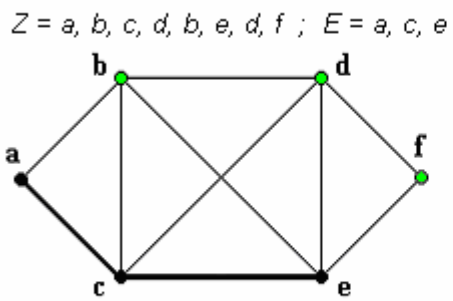


Obrázek 3.

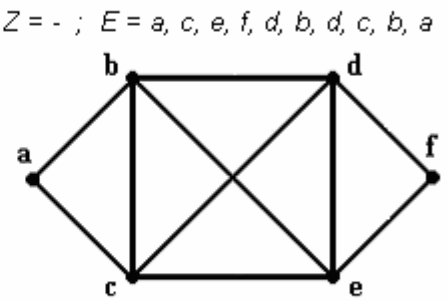


Obrázek 4.

Na poslední místo v zásobníku Z se tak dostává vrchol *e*, z kterého se dostaneme do vrcholů *d, f* a *e*, čímž jsme se opět dostali do slepé uličky. Vrchol *e* tedy přesuneme do E (obrázek 5). Jelikož jsme již vyčerpali všechny hrany můžeme postupně vyjímat jednotlivé vrcholy ze zásobníku Z a vkládat je do datové struktury E a postupně tak určovat E. Ve chvíli, kdy je zásobník Z prázdný dostáváme výsledný eulerovský tah, který má podobu *a, c, e, f, d, e, b, d, c, b, a* (obrázek 6).



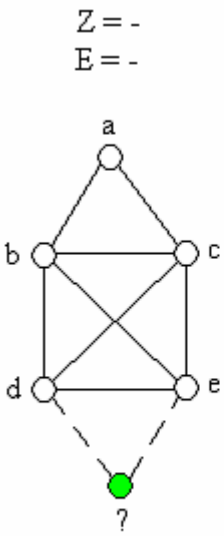
Obrázek 5.



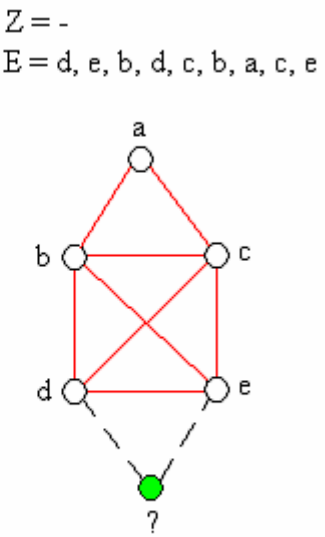
Obrázek 6.

Kreslení tahu v případě jednotažky

Při hledání tahu jednotažky postupujeme tak, že jednotažku změníme přidáním vrcholu (a k tomuto vrcholu dvou incidentních hran) na eulerovský graf, tzn. že do grafu vložíme pomocný vrchol, který si označíme například znakem „*?*“ a tento vrchol spojíme s vrcholy, které jsou právě lichého stupně (na obrázku 7 jsou nově vložené hrany zobrazeny přerušovaně). Pokud se jedná o sudý graf, pomocný vrchol nepřidáváme.



Obrázek 7.



Obrázek 8.

Při kreslení tahu pak použijeme předchozí algoritmus. Dostaneme tah *a, c, e, ?, d, e, b, d, c, b, a*. Z výsledného tahu odebereme přidaný vrchol ?, čímž z něj odebereme i přidané hrany. Výsledný tah je tvořen posloupností vrcholů a hran, začínající ve vrcholu *d* (tj. ve vrcholu, který následoval za přidaným pomocným vrcholem *?*) a končící ve vrcholu *e* (tj. ve vrcholu, který předcházel pomocný vrchol *?*). Pochopitelně dvojnásobný výskyt

krajního vrcholu *a*, který nám uzavíral eulerovský tah, uvažujeme jen jedenkrát. (obrázek 8).

Závěrem lze říci, že jednotažku můžeme kreslit i přímo, tj. bez přidání vrcholu *?*. Algoritmus však musíme začít v jednom ze dvou lichých vrcholů. Po prvním průchodu najdeme tah, který není uzavřený a končí v druhém z lichých vrcholů. Další nacházené tahy pak již budou vždy uzavřené.

8 Backtracking

Pro tento algoritmus jsme použili literaturu [1] a [3].

Jedná se o systematické prohledávání stavového prostoru a to *prohledávání do hloubky s návratem*[3]. Vždy, když se při prohledávání dostaneme k posloupnosti, která odpovídá úplnému řešení, provedeme test, zda toto řešení je přípustné. U optimalizační úlohy navíc zjišťujeme nejlepší dosud nalezené přípustné řešení. Základní princip backtrackingu spočívá v systematickém zkoumání všech potenciálních řešení. Algoritmus lze navrhnou asi takto:

1. Na začátku je částečné řešení prázdné.
2. Dosavadní částečné řešení je rozšířeno.
3. Pokud je nové částečné řešení kompletním řešením úlohy, skončí se.
4. Pokud nové částečné řešení vyhovuje podmínkám úlohy, jde se znovu na bod 2.
5. Pokud nové částečné řešení nevyhovuje podmínkám úlohy, je vyzkoušeno jiné.
6. Pokud žádné částečné řešení nevyhovuje podmínkám úlohy, vrátíme se o úroveň výš (na poslední vyhovující částečné řešení) a pokračujeme v prozkoumávání jeho dalších potenciálních rozšíření (bod 2.).

[3] Jedná se tedy o systematicky prováděnou metodu hrubé síly nebo též o zpětné sledování.

9 Metody analýzy kritické cesty

Pro seznámení se s touto kapitolou jsme použili literaturu [4] a [5].

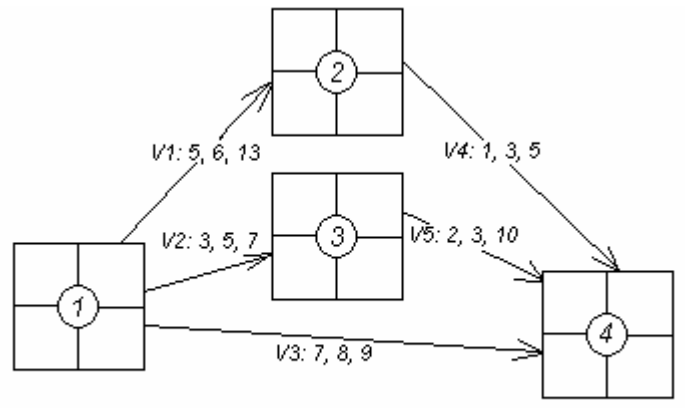
Představme si, že máme za úkol uskutečnit nějaký složitý projekt, který se skládá z mnoha dílčích činností , např. stavbu domu nebo továrny. Předpokládejme, že o každé dílčí činnosti předem víme, jak dlouho bude trvat nebo alespoň máme důvěryhodný odhad a známe seznam činností, které musí být ukončeny, aby tato činnost mohla začít. Např. než začneme stavět střechu domu, musí být hotové základy atd. Zajímá nás, kdy nejpozději budeme muset jednotlivé činnosti zahajovat, abychom neprodloužili dobu trvání celého projektu.

Takovéto situace popisujeme *síťovými grafy*, které jsou složeny z uzlů a hran. Hrany vyjadřují jednotlivé *dílčí činnosti* projektu a uzly představují body na časové ose, v nichž činnosti začínají a končí.

V přístupu k ohodnocení činností se rozlišují dvě nejstarší a nejjednodušší metody analýzy kritické cesty, které se značí zkratkami ***CPM*** (Critical Path Method) a ***PERT*** (Program Evaluation and Review Technik).

U metody CPM se předpokládá, že pro každou činnost známe přesně její odpovídající dobu trvání, kterou je možné např. vypočítat z rozsahu práce. Z hlediska ohodnocení hran je tedy metoda CPM deterministickým modelem skutečnosti.

Naopak je tomu u metody PERT, kde ohodnocení činností je náhodná veličina. Pokud by bylo možné takovou činnost mnohokrát opakovat formou nezávislých pokusů, mohli bychom výsledné doby jejího trvání popsat statistickým popisem, přesněji řečeno normální rozdělením. Jedná se tedy o stochastický model skutečnosti.



Příklad analýzy kritické cesty metodou PERT

Ohodnocení jednotlivých hran je popsáno třemi čísly, které vyjadřují odhady doby trvání nákladů apod. Jde o *optimistický (a)*, *normální (m)* a *pesimistický (b)* odhad.

Optimistický odhad odpovídá nejkratšímu možnému trvání činnosti za mimořádně příznivých podmínek a analogicky pro zbylé dva odhady. Z uvedených tří odhadů potřebujeme pro každou činnost sítě určit průměr a rozptyl, což vlastně znamená určit střední dobu trvání činnosti a pravděpodobnost splnění dané činnosti.

Střední doba trvání se značí *E(y)* a spočítá se:

$$E(y) = \frac{a + 4.m + b}{6}$$

Pravděpodobnost splnění dané činnosti se značí *D(y)* a spočítá se:

$$D(y) = \left(\frac{b - a}{6} \right)^2$$

Jakmile získáme na základě odhadů průměrné doby trvání jednotlivých činností, je další postup při hledání kritické cesty a časových rezerv činností naprosto shodný s metodou CPM[4], u níž pracujeme od počátku s jedinou dobou trvání činností. Metoda PERT ještě dodatečně umožňuje (po výpočtu časových rezerv a kritické cesty) zabývat se otázkami pravděpodobnosti splnění celé akce v termínech odlišných od doby trvání kritické cesty.

Vlastní výpočet kritické cesty je u obou metod stejný. Pracujeme se sítí a pro každou činnost máme jediný údaj o době trvání. U metody CPM pevný odhad (y_{ij}) a u metody PERT průměrný odhad doby trvání ($\overline{y_{ij}}$). Kromě těchto základních údajů známe ještě celkovou dobu trvání celého projektu nebo v jaké době bychom si přáli, aby celý projekt byl uskutečněn. Pro další veličiny, s nimiž budeme pracovat, si musíme zavést označení:

$t_i^{(0)}$	-	nejdříve možný začátek činnosti vycházející z uzlu *i*
$t_i^{(0)} + y_{ij}$	-	nejdříve možný konec činnosti *y*(*i*,*j*)
$t_j^{(1)}$	-	nejpozději přípustný konec činnosti vstupující do uzlu *j*
$t_j^{(1)} + y_{ij}$	-	nejpozději přípustný začátek činnosti *y*(*i*,*j*)

Kromě těchto základních veličin pro časové výpočty se užívají ještě další, které slouží ke zjišťování několika druhů časových rezerv.

[4] Vysvětlení, proč jsme situaci popisovali pouze na metodě PERT.

Celková časová rezerva se značí *CR*, spočítá se

$$CR = t_j^{(1)} - t_i^{(0)} - y_{ij}$$

a určuje, o kolik můžeme maximálně posunout dobu trvání činnosti, aby se nezměnil její nejpozději přípustný konec.

Volná časová rezerva se značí *VR*, spočítá se

$$VR = t_j^{(0)} - t_i^{(0)} - y_{ij}$$

a určuje nám, o kolik můžeme maximálně posunout dobu trvání činnosti, aby se nezměnil nejdříve možný začátek následující činnosti.

Nezávislá časová rezerva se značí *NR*, spočítá se

$$NR = t_j^{(0)} - t_i^{(1)} - y_{ij}$$

a určuje nám, o kolik můžeme maximálně posunout dobu trvání činnosti, aby se nezměnil nejpozději přípustný konec předcházející činnosti.

Závislá časová rezerva se značí *ZR*, spočítá se

$$ZR = t_j^{(1)} - t_i^{(1)} - y_{ij}$$

a určuje nám o kolik můžeme maximálně posunout dobu trvání činnosti, aby se nezměnil nejpozději přípustný konec předcházející činnosti.

Implementační část

Pořadí:	Uzel,	Min,	Max.
	1	0,00	0,00
	2	7,00	7,00
	3	5,00	6,00
	4	10,00	10,00

činnost,	čas.rezerva:	celková,	závislá,	nezávislá,	volná,
V1	-	0,00	0,00	0,00	0,00
V2	-	1,00	1,00	0,00	0,00
V3	-	2,00	2,00	2,00	2,00
V4	-	0,00	0,00	0,00	0,00
V5	-	1,00	0,00	0,00	1,00

N (10,00 ; 2,22)

Požadovaný termín lze splnit s pravděpodobností: 50,0000 %

Tento výpis jsme získali na základě předchozího obrázku. Je vidět pořadí jednotlivých uzlů a jejich nejdříve možné začátky (sloupec s názvem „Min“) a konce (sloupec s názvem „Max“) a nejpozději přípustné začátky a konce. Dále můžeme vidět jednotlivé dříve zmiňované časové rezervy pro jednotlivé uzly, konkrétní parametry náhodného rozdělení a dále s jakou pravděpodobností je „náš“ požadovaný termín splnitelný.

10 Implementace

Realizace zadání diplomové práce byla uskutečněna pomocí několika programů. Stěžejní program se jmenuje „*Vizualizace algoritmů v teorii grafů*“ a jedná se o aplikaci, která dovoluje „vizualizovat“ jednotlivé základní grafové algoritmy. Do této hlavní aplikace jsou ještě navíc implementovány některé další doplňující aplikace.

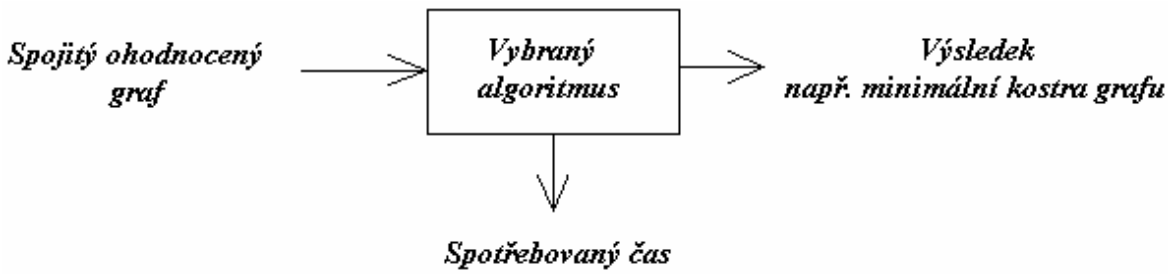
10.1 V jakém prostředí jsme programovali?

Celá aplikace byla vyvíjena v produktu zvaném *Delphi.* Právě pomocí vizuálního programovacího jazyka, jakým jsou Delphi, můžeme i složité a výkonné aplikace vytvářet snadno, rychle a efektivně, aniž by celá aplikace ztratila na svém profesionálním vzhledu.

10.2 Porovnání časových náročností

Mezi jeden z hlavních úkolů patřilo i zjištění časových náročností nejpoužívanějších algoritmů[5]. Abychom jednotlivé náročnosti mohli zjistit, bylo třeba nejdříve začít s měřením celkového času, který jednotlivé algoritmy potřebovaly, aby dospěly k určitým výsledkům.

Předpokládáme, že již máme vytvořený graf G, na kterém chceme zobrazit jednotlivé výsledky algoritmů.



Program v Delphi běží pod konkrétním operačním systémem. Pro měření času potřebného pro vykonání algoritmu jsem tedy naprogramoval programovou jednotku (*unit*), která splňuje naše požadavky.

[5] My jsme porovnávali algoritmy pro hledání minimální kostry grafu a prohledávání grafů.

Obsahuje dvě API funkce:

- *QueryPerformanceFrequency*, která vrací rozlišení tiků za sekundu a
- *QueryPerformanceCounter*, která vrací hodnotu čítače, s frekvencí QueryPerformanceFrequency

Tyto dvě API funkce lze aplikovat s velmi vysokým rozlišením, a to až na cca 1 mikrosekundu. Tedy při stisknutí požadovaného tlačítka pro vykonání algoritmu dojde ke spuštění časovače QueryPerformanceCounter (start) a současně při ukončení vykonávaného algoritmu se časovač zastaví, provede se rozdíl těchto dvou časů vydělený počtem tiků za sekundu a to následovně:

```
QueryPerformanceCounter(stop)
QueryPerformanceFrequency(frekvence)
KruskalGraf.Cas := ((stop-start)/frekvence)
```

Musíme si uvědomit, že moderní operační systémy jsou víceúlohové a víceuživatelské. Měřený program proto nikdy neběží sám a z tohoto důvodu se mohou doby samotného programu výrazně lišit. Před měřením časových závislostí jednotlivých algoritmů je tedy vhodné uzavřít všechny nepotřebné spuštěné aplikace, čímž ještě více zpřesníme výsledky. Aby byly výsledky věrohodnější, spustíme měřený algoritmus několikrát a jednoduše spočteme průměr.

Zjišťování časových náročností jednotlivých algoritmů budeme provádět na algoritmech, které patří v teorii grafů mezi nejpoužívanější, tj. jedná se o algoritmy pro hledání minimální kostry grafu a algoritmy sloužící k prohledávání grafů.

Dříve než začneme psát něco o složitostech algoritmů, měli bychom si říci, co to složitost vůbec je a jak se vyjadřuje.

10.3 Složitost algoritmu

Každá úloha má obvykle mnoho různých řešení, ale zřídkakdy jsou tato řešení všechna "stejně dobrá". Největší rozdíl mezi nimi (mimo komplikovanosti programu, samozřejmě) spočívá v času, za jaký se k výsledku doberou, a v množství paměti (prostoru), které na to spotřebují. Tyto vlastnosti se většinou vyjadřují ve formě časové a prostorové složitosti.

Složitost závisí na velikosti vstupních dat, proto ji můžeme popsat jako funkci *T(n)*, kde číslo *n* udává velikost vstupních dat. Například $T(n) = an + b$, kde *a*, *b* jsou nějaké konstanty, je zápis lineární časové složitosti (složitost roste lineárně s rostoucí velikostí vstupů). Obvykle je pro odhad složitosti důležitý pouze typ funkční závislosti a nikoliv přesné hodnoty konstant - ty se zanedbávají.

Za **efektivní algoritmy** považujeme takové postupy, jejichž složitost je polynomiální (např. n^{127}), nikoliv exponenciální (např. 2^n). Provádění exponenciálních algoritmů či dokonce algoritmů o složitosti $T(n) = n!$ může už jen při malém navýšení velikosti vstupních dat trvat i mnoho tisíců a miliónů let. Čím je algoritmus složitější, tím menší je vliv navýšení výkonu procesoru při velké velikosti vstupu.

V zápisu $T(n) = an + b$ je *a* multiplikativní konstanta, která v sobě zahrnuje počet operací ve strojovém kódu, které je třeba provést pro vyřešení jedné operace na úrovni vyššího programovacího jazyka. Aditivní konstanta *b* udává pouze konstantní nárůst složitosti nezávislý na velikosti vstupních dat. Velikost této konstanty závisí jen na daném počítačovém systému, který algoritmus provádí. Při značné velikosti vstupních dat (to je situace, která nás vzhledem k složitosti algoritmu zajímá, při malých vstupech je složitost všech algoritmů poměrně nízká a vyrovnaná) můžeme konstanty *a* a *b* zanedbat, protože vzhledem k velikosti *n* jsou nepatrné, a výsledná složitost *T(n)* tak závisí především na velikosti *n* - velikosti vstupních dat algoritmu. Vybereme-li ze všech konstant *a* závisejících na použitém kompilátoru, který překládá program, největší konstantu, kterou označíme *c*, platí $T(n) \leq cn$ pro všechna dostatečně velká *n*. Tuto skutečnost značíme $T = O(n)$, čímž vyjadřujeme asymptotické chování funkce *T*. Z této úvahy vychází matematická definice pojmů horní a dolní odhad složitosti algoritmu a složitost v průměrném případě (očekávaná složitost).

Horní odhad složitosti algoritmu nás zajímá nejvíce, protože nám udává složitost algoritmu v nejhorším případě. Ovšem algoritmus dosahuje této horní složitosti jen ve vzácných případech, v takovém případě pak o složitosti algoritmu vypovídá lépe složitost průměrná.

Pro názornost uvedeme přiklad na zjištění prostorové složitosti:

Pro každý z N vrcholů potřebujeme 4 integery a 1 pointer, vše po 4B, pro každou z M hran 2 integery po 4B. K tomu nějakou konstantně velkou paměť. Skutečné nároky jsou tedy 20*N + 8*M + ta konstanta. V asymptotickém vyjádření to bude O(M+N). Aditivní a multiplikativní konstanta se nepočítají.

Z aplikace vyplívá, máme – li definované třídy (viz Příloha 1) pro uzly, hrany atd., pak zabírají asi následující část paměti:

	objekt	typ	paměť
TGrafUzel	FNumber	integer	4 B
	FFlag	boolean	2 B
	FPredek	seznam	dle počtu uzlů min = 1 uzel (10 B) max = 1499 uzlů (14990 B)
	FNaslednik	seznam	dle počtu uzlů min = 1 uzel (10 B) max = 1499 uzlů [6] (14990 B)
	FVaha	integer	4 B
celkem			min = 20 B [7] max = 15000 B

	objekt	typ	paměť
TGrafHrana	FVaha	integer	4 B
	FFromU	TGrafUzel	dle počtu uzlů min = 1 uzel (10 B) max = 1499 uzlů (14990 B)
	FToU	TGrafUzel	dle počtu uzlů min = 1 uzel (10 B) max = 1499 uzlů (14990 B)
celkem			min = 24 B [8] max = 29984 B

10.4 Časové náročnosti algoritmů

Každá z těchto metod pracuje s jinou časovou náročností a více či méně se hodí na určitý typ grafů.

Název algoritmu	Časová náročnost

[6] Liší se v závislosti na velikosti paměti počítače.
[7] 10 B zabírá samotný uzel a 10 B zabírá buď následník nebo předek při minimálním počtu uzlů 2.
[8] Hrana obsahuje dva uzly (předek, následník), celkem tedy minimálně zabírá 10 + 10 + 4 = 24 B.

Kruskalův algoritmus	O (m.log n)
Jarník – Primův algoritmus	O (n^2)
Borůvkův algoritmus	O (m.log n)
Prohledávání grafu do šířky	O (m+n)
Prohledávání grafu do hloubky	O (m+n)

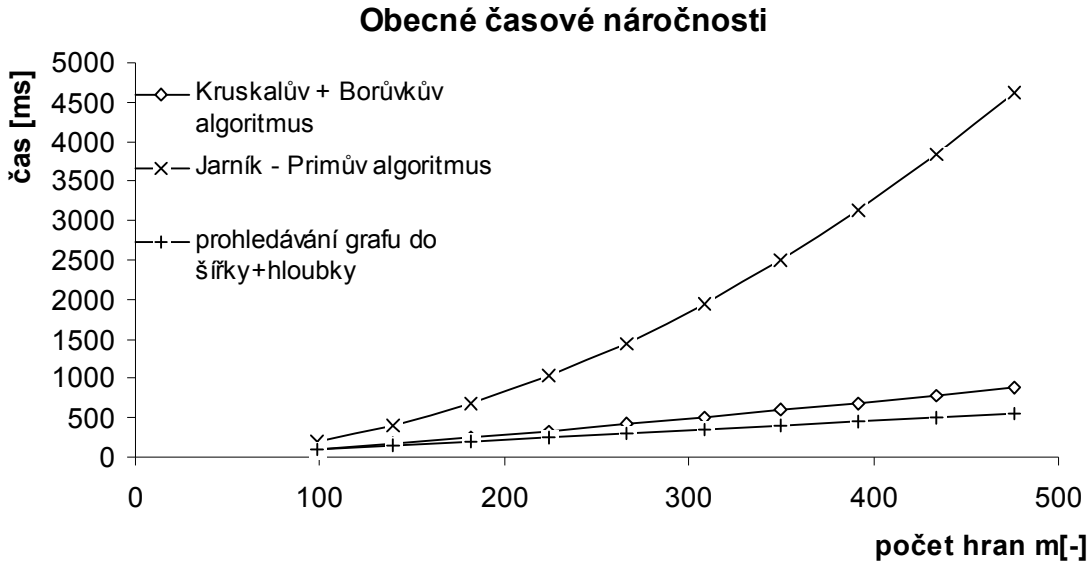
Tabulka 1. Časové náročnosti jednotlivých algoritmů

Odvození, jak bylo dosaženo předešlých časových náročností, je uvedeno na straně 15.

Málokdy se setkáme s úkoly počítat minimální kostry z úplných grafů. Mnohem častěji se setkáme s reálnými grafy, jejichž počet hran je nějakým násobkem počtu uzlů. Pro následující grafy byly generovány grafy s *n* uzly a *m* = *7*n* hranami.

Pozn. Ve skutečnosti nám ani moc nejde o to, jaký typ grafu budeme zkoumat, ale o jednotlivé charakteristiky.

počet hran m [-]	**Kruskalův + Borůvkův alg.** O (m.log n)	**Jarník – Primův alg.** O (n^2)	**prohledávání grafu do hloubky + šířky** O (m+n)
98	112,320	196	112
140	182,144	400	160
182	257,525	676	208
224	337,154	1024	256
266	420,222	1444	304
308	506,183	1936	352
350	594,639	2500	400
392	685,289	3136	448
434	777,898	3844	496
476	872,274	4624	544



V následujících grafech a tabulkách se setkáme s naměřenými časovými hodnotami jednotlivých algoritmů. Pro každý algoritmus je vždy uveden jeden graf. Popisují závislost poměru naměřeného času a obecné náročnosti algoritmu na celkovém počtu hran. První křivka (ta spodní) popisuje naměřenou časovou náročnost algoritmu.

Druhá křivka znázorňuje obecnou časovou náročnost algoritmu přepočítanou k první naměřené hodnotě, tj. spočítáme poměr naměřeného času prvního grafu s obecnou časovou náročností tohoto grafu (např. pro Kruskalův algoritmus: $k = t / m.\log n$). Tímto podílem získáme časovou konstantu, se kterou pak násobíme každou obecnou časovou náročnost. Podstatné je to, že křivka naměřené časové náročnosti nesmí v žádném svém bodě překročit obecnou časovou náročnost přepočítanou k prvnímu měřenému bodu. Kdyby překročila, znamenalo by to, že jsme algoritmus naprogramovali nekorektně.

Při měření jednotlivých časových náročností jsem pro větší objektivnost výsledků podroboval jednotlivé vygenerované grafy algoritmům a postup opakoval pětkrát.

n [-]	m [-]	t [s]	m.log n [-]	t / m.log n [s]	t [s]
5	10	0,000034	6,99	0,00000495	0,000035
8	28	0,000056	25,29	0,00000222	0,000125
10	45	0,000116	45,00	0,00000259	0,000223
100	4950	0,0066	9900,00	0,000000667	0,049005
200	19900	0,033	45990,49	0,000000732	0,226663
500	124750	0,26	336696,51	0,000000773	1,666650
1000	499500	1,25	1498500,00	0,000000834	7,417575
1200	719400	1,79	2215162,99	0,000000809	10,965056
1500	1124250	3,01	3570720,60	0,000000844	17,675067

Tabulka 2. Získané hodnoty pro Kruskalův algoritmus

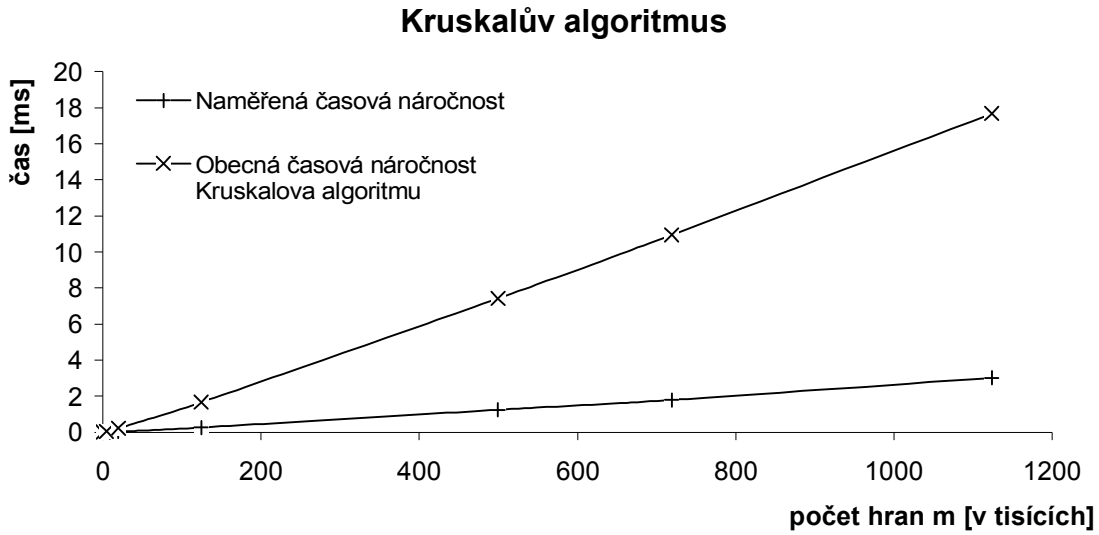
Legenda: n ..... počet uzlů [-]

m .... počet hran [-]

t ..... naměřená časová náročnost [s]

m.log n ..... konkrétní časová náročnost [-]

t / m.log n [s] ..... poměr naměřeného času s obecnou časovou náročností



Zbylé tabulky a příslušné grafy jsou uvedeny v Příloze 2.

10.5 Zhodnocení dosažených výsledků

Z výše uvedené tabulky 2 a z tabulek uvedených v příloze jsou jasně vidět jednotlivá asymptotická chování poměru *t/náročnost*. Všechny tyto algoritmy jsme testovali na třech různých typech grafů:

- malé grafy (5 – 10 uzlů)
- střední grafy (100-500 uzlů)
- velké grafy (1000-1500 uzlů)

s výjimkou Jarník – Primova algoritmu, který jsme testovali na grafech pouze do tisíce uzlů, protože jsme využili maticové reprezentace grafů[9]. Na druhé straně se nám matici podařilo nadefinovat právě pouze do tisíce uzlů z důvodu nedostatku paměti. Všechny tyto grafy jsou vygenerovány jako úplné, které mají všechny hrany ohodnoceny stejným číslem a to číslem jedna. Je to kvůli lepšímu přizpůsobení k programu Mathematica 4.0, který byl zapůjčen katedrou aplikované matematiky TUL a s kterým byly jednotlivé „časy" algoritmů porovnávány.

	Kruskalův algoritmus			**Jarník – Primův algoritmus**			**Borůvkův algoritmus**		
počet uzlů n [-]	*5*	*8*	*10*	*5*	*8*	*10*	*5*	*8*	*10*
Mathematica 4.0 [s]				0,01	0,02	0,03			
doba trvání algoritmu [μs]	34,6	56	116	34,1	67,4	79,2	111	137	472
doba trvání algoritmu i s vykreslením výsledků [μs]	34,6	56	116	34,1	67,4	79,2	111	137	472

Tabulka 3. Získané výsledky pro malé grafy

Jak je z tabulek 3 - 5 vidět, prováděl jsem tři různá měření a zaznamenával jejich výsledky. Jelikož program Mathematica 4.0 neumožňuje výběr některého z algoritmů, z výsledného tvaru grafu jsem usoudil, že konkurenční program využívá pro určení minimální kostry grafu Jarník – Primův algoritmus (proto příslušná políčka zůstala

[9] Přesněji řečeno: použili jsme matici sousednosti.

nevyplněna). Jednotlivé výsledky v posledních dvou řádcích tabulky 3 jsou stejné, protože aplikace neumožňuje měření algoritmů i s vykreslením. Jedná se stejně o tak nepatrné výsledky, že při přesném odměřování obou by šlo o zanedbatelné rozdíly.

	Kruskalův algoritmus			**Jarník – Primův algoritmus**			**Borůvkův algoritmus**		
počet uzlů n [-]	*100*	*200*	*500*	*100*	*200*	*500*	*100*	*200*	*500*
Mathematica 4.0 [s]				0,91	3,24	18,67			
doba trvání algoritmu [ms]	6,6	33,5	260	4,66	8,79	82,65	7,23	293	480
doba trvání algoritmu i s vykreslením výsledků [s]	0,6	1,1	4,7	0,5	0,9	1,3	0,6	1,1	4,5

Tabulka 4. Získané výsledky pro středně velké grafy

	Kruskalův algoritmus			**Jarník – Primův algoritmus**	**Borůvkův algoritmus**		
počet uzlů n [-]	*1000*	*1200*	*1500*	*1000*	*1000*	*1200*	*1500*
Mathematica 4.0 [s]				73,13			
doba trvání algoritmu (1) [s]	1,25	1,79	3,02	0,562	4,32	7,42	13,46
doba trvání algoritmu i s vykreslením výsledků (2) [s]	26,9	49,2	138,1	18,6	43,2	52,4	154,6

Tabulka 5. Získané výsledky pro velké grafy

Vzhledem k tomu, že jak v programu Mathematica, tak i v naší aplikaci při testování velkých grafů, tedy grafů s 1500 uzly a více, oba programy nebyly schopny vzhledem k zaplnění operační paměti test dokončit a dále vzhledem k praktické využitelnosti grafů

s tak velkým počtem uzlů, omezil jsem se pouze na testování grafů s výše uvedeným počtem uzlů.

Všechny výsledky byly pořízeny na PC Celeron 633 Mhz. Tyto údaje uvádím proto, že časové výsledky získané z jiných počítačů budou lepší či horší (závisí na konfiguraci počítače), ale rozhodně budou ve stejném poměru.

Závěry vycházející z těchto tabulek lze rozdělit do dvou skupin dle zaměření jednotlivých subjektů (lidí) na závěry pro vědecké účely a závěry pro „běžného“ uživatele. Závěry pro vědecké účely jsou smysluplné spíše pro lidi, kteří se zabývají vědeckou stránkou těchto problémů, s největší pravděpodobností je asi budou zajímat jednotlivé časové náročnosti algoritmů, tedy jejich efektivnost (viz 1 v tabulce 5), zatímco okruh zájmu „běžných“ uživatelů je zřejmě takový, že je bude zajímat čas, kdy budou vidět výsledky jednotlivých algoritmů (viz 2 v tabulce 5).

Co se týče porovnání jednotlivých časových náročností algoritmů našeho a konkurenčního programu, náš algoritmus je u malých a středně velkých grafů řádově třistakrát rychlejší, kdežto u grafů velkých je rychlejší řádově stokrát.

Zaměříme – li se na výsledky získané pouze z naší aplikace, zjistíme, že co se týče malých grafů, vycházejí výsledky u Kruskalova a Jarník – Primova algoritmu naprosto srovnatelné. Borůvkův algoritmus je nejpomalejší. U středních grafů se již rozdíly samozřejmě projevují více. Pořadí je následující: Jarník – Primův, Kruskalův a Borůvkův algoritmus. V rozmezí uzlů 1000 a více jsou výsledky ve stejném sledu a téměř zachovaném stejném poměru.

Závěrem však je třeba zdůraznit, že použitím jiných datových struktur v jednotlivých algoritmech můžeme dosáhnout lepší či horší efektivnosti algoritmů, tedy lepších či horších časových náročností.

10.6 Úloha obchodního cestujícího

Jsou zde implementovány tři algoritmy "spojování bodů" v jeden graf:

- **Prostá cesta** - spojuje v libovolném pořadí body s požadavkem, aby se jednotlivé čáry nekřížily.
- **Konvexní obal** - umí malovat minimální délku "plotu" kolem nasbíraných bodů s vlastností, že všechny vnitřní úhly jsou menší než 180^{o}. Jestliže body jsou kolíky, pak provázek obtočený kolem těchto kolíků definuje konvexní obal.

- **Nejkratší cesta** představuje problém obchodního cestujícího. Cestující potřebuje navštívit množinu měst a chce plán své cesty takový, aby jeho cesta (co do vzdálenosti) byla tak krátká jak jen to je možné.

Prostá cesta

Technika pro řešení tohoto problému je prostá. Začínáme v bodě, který má jednu ze souřadnic maximální (v tomto případě maximum y-ové hodnoty). Bereme v úvahu čáry vytvořené spojením, které ukazují na další bod v množině. Počítáme úhly od horizontály každého bodu a třídíme je podle úhlu. Cesta z počátečního bodu do každého dalšího vytříděného bodu v posloupnosti a zpátky do počátečního bodu tvoří prostou uzavřenou cestu.

Konvexní obal

Vezměte bod (s největší hodnotou y) a vypočítejte vnitřní úhel z tohoto bodu do všech ostatních bodů – vyberte bod s největším vnitřním úhlem (nebo nejmenším úhlem od horizontály), nakreslete čáru do tohoto bodu. Myšlenka je taková, že uděláme jeden kompletní oběh, když přidáme tyto obalové body (když spojíme první obalový bod s druhým atd., vznikne nám obal). Opakujte z již přidaného bodu s dalším omezením, že nový bod je ten s nejmenším úhlem, který je zároveň větší než předchozí úhel. Pokračujte, dokud se nedostanete do počátečního bodu.

1. Nakreslete několik bodů na kousek papíru a vyberte nejnižší z nich, P1.
2. Nakreslete polopřímku z tohoto bodu pokračující směrem doprava a položte vaši tužku na tuto linku gumou na P1.
3. Nechte gumu na P1 (nebo Pn) a otáčejte tužkou proti směru hodinových ručiček dokud se nedotknete bodu. To je další bod obalu, P2 (Pn+1).
4. Nakreslete čáru z P1 do P2 (z Pn do Pn+1) a posuňte tužku podél této linky dokud se guma nedostane na bod P2 (Pn+1).
5. Opakujte kroky 3 a 4, dokud se nedostanete zpět do bodu P1 a obal je kompletní.

"Klouzání tužky" nám umožňuje posouvat se po obalu. Programovým ekvivalentem je při prohledávání ukládat každý úhel do proměnné a ignorovat všechny úhly menší než úhel v této proměnné. Vybereme nejmenší úhel větší než předchozí a bod, pro který jsme tento úhel zjišťovali, je dalším bodem obalu.

Nejkratší cesta

Jedná se o nejširší třídu problémů nazývaných NP úplné úlohy (nedeterministické polynomiální úplné). Tento termín odpovídá skupině problémů pro které není známo polynomiální časové řešení, ale nebylo dokázáno, že neexistuje žádné řešení. Úlohy, které mohou být řešeny v polynomiálním čase, mají časy řešení úměrné kombinacím mocnin počtu prvků. Úlohy, které nejsou polynomiální, jsou exponenciální (nebo horší) a časy řešení rostou úměrně s funkcemi počtu prvků jako mocninou (např. 2^N). Úplná vyhledávání požadující N! operací jsou jedny z „horších“ případů.

Verze obsažená zde zkouší hrubou sílu, úplnou techniku vyhledávání kontrolováním délek cest pro každé možné pořadí navštěvovaných měst. Může zkontrolovat 50000 až 200000 cest za sekundu a pracuje spolehlivě až do deseti bodů (3,6 mil. zkontrolovaných cest). Algoritmus není vhodný pro řešení případu patnácti měst a více (okolo trilionu spojení, 10^{12} cest). Jednoduchým setříděním bodů tak, že každý je vedle svého nejbližšího souseda, můžeme obvykle dostat cesty, které dávají docela dobré výsledky v rozumném čase.

Tato sekce programu užívá **TComboSet** třídu definovanou v **Combo** unitu. TComboSet poskytuje logiku potřebnou ke generování všech permutací počtu bodů k testování všech možných cest.

Úloha obchodního cestujícího je problémem o nalezení nejkratší Hamiltonovské kružnice v úplném neorientovaném grafu, jehož hrany jsou ohodnoceny délkami.

10.6.1 Úloha obchodního cestujícího

Tato úloha je považována za jeden z nejtěžších řešených algoritmů a patří do kategorie NP - úplných úloh. Situace v detailech je následující :

Formuluje se tak, že obchodní cestující má za úkol navštívit n měst v libovolném pořadí a vrátit se zpět tak, aby jeho trasa byla co nejkratší. Přitom se předpokládá, že vzdálenosti mezi všemi dvojicemi měst jsou předem známé a symetrické (v obou směrech je vzdálenost stejná) a že pouze jedna cesta může spojit dvě města a žádná jiná (graf je prostý).

Pro tento problém jsou implementovány dva algoritmy. Jeden je *heuristický,* druhý se nazývá *exhaustivní (úplný).* Heuristický algoritmus nedává vždy nejlepší řešení, ale na druhou stranu je velice rychlý a nenáročný na zdroje. Oproti heuristickému algoritmu nám úplný algoritmus dává vždy nejlepší řešení, ale naopak je velice pomalý a výpočtově náročný. Např. máme-li jako vstup 20 měst a spustíme úplný algoritmus na (v současné době) běžném počítačovém systému, bude to trvat cca. 1 mil. století, než nám dá

výsledek. Ve skutečnosti (prakticky) zjišťujeme, že pro vstup $n \geq 10$ je algoritmus velice pomalý.

Analýza heuristického algoritmu

Jak je uvedeno výše, tento algoritmus se snaží rychle se blížit k nějakému řešení raději než k řešení optimálnímu. V některých případech je možné, že nám dá řešení velmi vzdálené od optimálního. Jeho základní předností je, že je velice rychlý. Základní idea algoritmu je taková, že začínáme ze základního města a po kalkulaci jeho vzdáleností s jeho sousedícími městy vybereme město s minimální vzdáleností. Potom pokračujeme stejným způsobem pro další město a všechna ostatní, dokud nenavštívíme všechna města. Měli bychom zdůraznit, že protože jsme předpokládali, že město nemusí být spojeno s jiným, tento algoritmus může skončit ve slepé uličce, v tomto případě končí neúspěšně.

Předpokládejme, že obchodní cestující má navštívit n měst. Pro první město algoritmus vytváří (n-1) porovnání. Po výběru druhého města vytváří (n-2) porovnání (máme již vybráno jedno a nemůže být vybráno znovu) a tak pokračujeme, dokud nedosáhneme (n -1) měst, kde algoritmus končí.

Můžeme říci, že složitost algoritmu je: $\delta_n = (n-1) + (n-2) + (n-3) + \ldots + 0 = \mathbf{(n^2 - n)/2}$.
A jak můžeme vidět, složitost je polynomiální $\mathbf{O(n^2)}$.

Analýza úplného algoritmu

Zde začínáme ze základního města a zkoušíme všechny možné cesty, které existují. Kalkulujeme jejich váhy a vykazujeme cestu s nejnižší váhou. Je zřejmé, že číslo možných cest je číslo permutací n měst, která je (n - 1)!. Zjišťujeme také, že pro každých (n - 1)! porovnání uděláme přibližně 2n operací.

Můžeme říci, že složitost algoritmu je: $\delta_n = (n-1)!(2n) = \mathbf{2n!}$.
A jak můžeme vidět, složitost je opět polynomiální **O (n!).**

Praktické výsledky (pouze pro některá N):

	HEURISTICKÝ ALG.			EXHAUSTIVNÍ ALG.		
N	**Počet kroků**	**Čas (sec)**	**Cena**	**Počet kroků**	**Čas (sec)**	**Cena**
4	12	0,0032	1640	36	0,0039	1640
5	20	0,0032	2741	253	0,0100	2741
6	30	0,0027	1889	1547	0,0445	1889
7	42	0,0036	2335	10065	0,2732	2335
8	56	0,0040	3333	85663	2,0560	3333
9	72	0,0044	3033	637180	18,1300	3033

Jak můžeme vidět, pro heuristický algoritmus počet kroků a uplynulý čas jsou na velmi nízké úrovni. Naproti tomu výsledky úplného algoritmu vzrůstají velice rychle[10]. Prakticky pro hodnoty větší než devět je metoda velice pomalá a tudíž téměř nepoužitelná. Je nutné dodat, že získané výsledky mohou kolísat na různých počítačích.

10.7 Problém čtyř barev

Na úvod poznamenejme, že pro tři a více barev žádný efektivní postup „barvení map“ není znám. Naopak, rozhodnutí, zda graf je uzlově r – barevný pro $r \geq 3$, je NP – úplnou úlohou a neexistuje pro něj žádný efektivní algoritmus (důkaz viz. [3]).

Dále uvedený postup pro barvení grafu nejmenším možným počtem barev je klasický příklad backtrackingu.

Jak funguje backtracking?

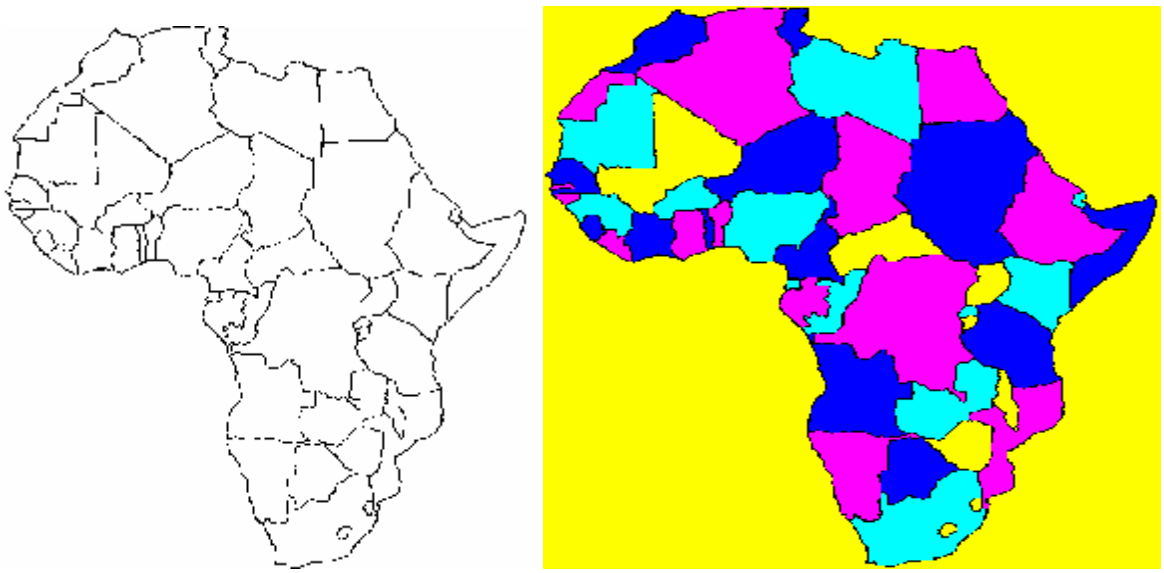
Při postupu vpřed, tj. při prodlužování částečné posloupnosti barev, dáme následujícímu vrcholu x jako výchozí vždy nejnižší barvu, která (momentálně) není použita u vrcholů z množiny $P(x)$, kde $P(x) = V(x) \cap \{1,2,...,x\text{-}1\}$ a x je vrchol. Nižší barvu nelze použít (při stávajících barvách předchozích vrcholů), vyšší barvy budou zkoušeny v dalším postupu backtrackingu. Návraty v backtrackingu budeme provádět při dosažení úplného obarvení.

[10] Pro porovnání jsem zkoušel obě metody na počítači s procesorem AMD 800 MHz.

Cílem backtrackingu je snížit barvu ve vrcholu *y*. Poněvadž však nižší barvy v *y* již jsou vyzkoušeny, je třeba změnit (tedy zvýšit) barvu v některém předchozím vrcholu *y*. Aby to však mělo vliv na vrchol *y*, musí být $x \in P(y)$, proto provedeme prvý návrat až do vrcholu *x* a v tomto vrcholu se pokusíme zvýšit barvu. Nelze – li již barvu ve vrcholu *x* zvýšit, provedeme další návrat do vrcholu *x-1*, *x-2* atd.

Implementační část

Jako vstup zde figuruje pouze černobílá bitmapa, která se načte a zobrazí na monitoru. Na tyto bitmapy může být aplikováno *hladové* a *minimální* obarvení. Jak již jsme říkali, hladový algoritmus se snaží velice rychle dospět k nějakému řešení raději, než k řešení optimálnímu. Výstupem *hladového obarvení* je obarvení celé mapy a to právě pěti barvami. Poté můžeme na takto obarvenou mapu aplikovat algoritmus pro hledání *minimálního obarvení*[11]. Zjistíme, že dojde k obarvení již menším počtem barev a to právě čtyřmi.



Na obrázku můžeme vidět, jak z neobarvené mapy vznikla mapa obarvená právě čtyřmi barvami.

[11] Byl implementován backtracking.

10.8 Aplikace „Graf skrz“

Najde nejkratší a nejdelší cestu skrz mřížku (na obr. níže) s čísly z bodu "S" do bodu "F". Cesta může být přímá nebo diagonální směrem vpravo, ale nikdy ne směrem doleva.

Není znám žádný způsob, jak vyřešit tento problém bez procházení všech cest a vybrání těch, které mají největší nebo nejmenší součet ohodnocení.

V jednoduchém pojetí je graf množina bodů nebo čísel (zvaných vrcholy) a „linek“ sloužících k jejich spojování (zvaných hrany). Labyrinty (bludiště) jsou grafy, kde každý „čtvereček“ je vrchol a volné cesty pro další pohyb jsou hrany (náš případ).

Implementační část

Zde je použita prohledávací technika a to "prohledávání do hloubky". Prochází každou cestu tak rychle, jak je to možné, než zkusí další.

Rekurzivní volání vyrobené procedury GetPath kontroluje uzly, které jsou nahoře vpravo, přímo rovně a vpravo dole - tedy tři možné cesty z každého uzlu. Komponenta *StringGrid* je použita pro zobrazení řešení. Možnost volby změny velikosti herního plánu je poskytována metodou *OnDrawCell* v komponentě *StringGrid* k animování vyhledávacího algoritmu – je tedy možné sledovat cestu, která je právě testována. Hranice představuje dvourozměrné dynamické pole čísel typu integer. Hranice má rozměr herního plánu + 2 pixely v každém směru, takže můžeme dodržovat přesnou hranici kolem čísel, abychom jednoduše testovali, kdy dosáhneme hrany.

			21			
		03	22	01		
	06	24	05	01	10	
S	22	07	11	24	08	F
	20	20	15	04	01	
		10	14	09		
			07			

Zkoušené cesty: 141

Max. délka cesty: 22 24 11 24 10 , Součet = 91

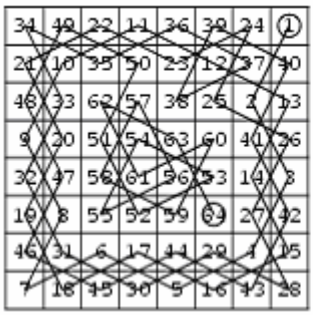
Min. délka cesty: 6 3 5 1 8 , Součet = 23

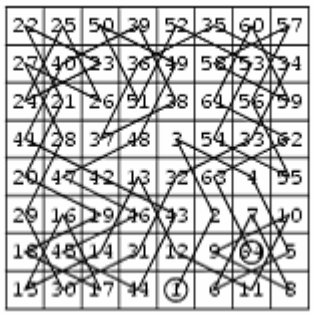
10.9 Rytířova cesta

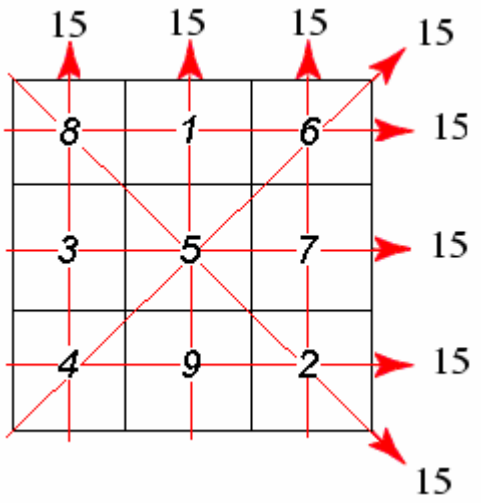
1	60	15	24	47	36	13	26
16	23	64	59	14	25	38	35
63	2	61	46	37	48	27	12
22	17	56	49	58	51	34	39
3	62	21	52	45	40	11	28
18	55	44	57	50	31	8	33
43	4	53	20	41	6	29	10
54	19	42	5	30	9	32	7

„Rytířova cesta“ je klasický šachový problém, který byl studovaný (a pravděpodobně vyřešený) již před 1000 lety. Matematik Euler publikoval první matematickou analýzu problému v roce 1759.

Připomínám, že „rytíř“ (kůň) se může pohybovat do tvaru L, takže se přemístí o jedno políčko v jednom směru a o dvě ve směru kolmém. Nyní nechť se rytíř může pohybovat na šachovnici o rozměru n x n, jejíž políčka jsou očíslována od 1 do n^2. Dráha se nazývá cesta, pokud rytíř navštíví každé pole právě jednou. Diagramy uvedené níže znázorňují tři rytířovy cesty na šachovnici o rozměru 8 x 8 [8].







Pole n x n čísel od 1 do n^2 je nazýváno magické, pokud součet čísel v každém řádku, sloupci a diagonále je číslo známé jako magická konstanta čtverce. Například na obrázku uvedeném výše je magická konstanta rovna 15. Čtverec, který není plně magický pouze

proto, že součet nedává na jeho diagonálách dohromady magickou konstantu čtverce (ale součty ve sloupcích a řádcích již konstantu dají) se nazývá polomagický čtverec. Magickou rytířovou cestou nazveme takovou cestu, která když jednotlivé skoky koně očíslujeme, vyplní šachovnici do podoby magického čtverce.

Je dlouho známo, že magické rytířovy cesty nejsou možné pro desky o rozměru *n x n*, pro *n* liché. Je také známo, že takové cesty jsou možné pro desky o velikosti *4k x 4k*, pro $k \geq 2$. Jakmile byl znám počet polomagických cest rytíře na obyčejné šachovnici 8 x 8, nebylo známo, zda existují nějaké plně magické cesty na stejné ploše.

Jaké jsou dosažené výsledky?

Tento dlouhodobý otevřený problém byl nyní vyřešen jako negativní. Po 61,40 CPU-dní, odpovídajících 138,25 dní výpočtu na PC o 1 GHz byl projekt dokončen 5.8. 2003. Navíc k síti celkových 140 odlišných polomagických cest rytíře výpočet poprvé demonstroval, že žádná magická cesta rytiře není možná, což nakonec tento dlouho otevřený problém pohřbilo [8].

Implementační část

Tuto úlohu lze chápat tak, že z libovolné startovní pozice má šachová figurka koně projít všechna políčka šachovnice tak, aby každé navštívila právě jednou.

Tento program je logickým rozšířením techniky uváděné v programu *Graf Traverse*. Nejdříve budeme vykonávat rekurzivní prohledávání do šířky voláním funkce *Res*. Po každém úspěšném pohybu zavoláme proceduru *Res* s novou polohou. Poté, co jsme figurku přesunuli 64 krát (počáteční umístění koně je pohyb číslo 1), máme problém vyřešen.

Úspěšný tah je tah do políčka, které nebylo ještě předtím navštíveno. Na rozdíl od programu *Graf Traverse* úplné vyhledávání nebude potřebné v případě, že potřebujeme najít pouze jednu cestu, která projde všemi políčky na šachovnici. Složitost je v tom, že všech možných cest je daleko více než v programu Graf Traverse. Pro každý pohyb jsou možnosti volby mezi 0 a 8 pro další pohyb. Jestliže předpokládáme dva možné pohyby pro každou pozici, bude tam 2^{63} (okolo 10^{20}) cest. Při rychlosti prohledávání milionu cest za sekundu bychom potřebovali několik milionů let, abychom nalezli řešení!

Naštěstí je pomoc při ruce. Technika známá jako *Warnsdorfský heuristický* algoritmus nám dovoluje o mnoho lepší volby výběru dalšího pohybu než náhodný výběr. Tento algoritmus (objevil H. C. von Warnsdorf v roce 1823) přikazuje vybrat jako náš další pohyb

ten, který má nejméně možností dalšího skoku. Tento algoritmus pracuje tak, že protože máme implementován backtracking k odstranění špatných voleb pohybů, nemůžeme vidět pohyby "stažené zpět" při tvoření kompletní cesty.

Funkce *Res* počítá**,** kolik dalších pohybů by mohlo být přípustných pro každý platný potenciální pohyb z aktuální pozice. Tento seznam potenciálních pohybů je tříděn podle počtu dalších pohybů a nejmenší je vybrán. Pohyb je poté uskutečněn a rekurzivním voláním funkce *Res* je "vyrobena" další pozice. Jestliže volání nám vrátí „pravdu“ (další pozice je platná), opustíme pohyb v místě a ukončíme s výsledkem pravdy. Jestliže funkce zkolabuje (vrátí false), pohyb je stažen zpět a je zkoušen další potenciální pohyb. Jestliže již není žádná další cesta ke zkoušení, skončíme neúspěšně.

TBoard je objekt odvozený z komponenty *TStringGrid* a ovládá jak výpočtové tak i zobrazovací aspekty problému. *TBoard* tedy obsahuje výpočetní rutiny jako *platny_pohyb, ZobrazPohyb, pohyb_zpet, Res***.** Byla ještě přidána komponenta, která dovoluje uživateli měnit za běhu rychlost řešení a tedy i zobrazování výsledků.

10.10 Skladník

Tuto úlohu jsme od začátku vytvářeli jako úlohu, kde máme za úkol najít co nejdelší cestu v grafu.

Hala, v které jsou skladové prostory, má obdélníkový půdorys velikosti *m* x *n* jednotkových čtverců. V této hale se nacházejí různé skříně, police a jiné překážky, které vždy zabírají nějakou obdélníkovou oblast v základní síti haly. Podnik koupil čistícího robota, který má obcházet halu a přitom čistit neobsazené časti haly. Jelikož robot je citlivý na vlhkou dlažku, kterou po sebe zanechává při čištění, v podniku se rozhodli, že když se tato porucha neodstraní, pokusí se pro robota navrhnout takovou trasu, aby vyčistil maximální prostor a přitom dvakrát nevstoupil na ten stejný čtverec.

Implementační část

Program nejprve načte a vykreslí halu na obrazovku tak, že se nakreslí čtvercová síť (velikost jednotkového čtverce je konstanta) a obsazená políčka se vybarví tmavošedou barvou.

Trasu pro robota hledáme metodou prohledávání s návratem (backtracking), přičemž průběh algoritmu znázorňujeme na obrazovce tečkami uprostřed čtverců.

Labyrint v programu reprezentujeme pomocí neorientovaného grafu, v kterém každý čtverec je jeden vrchol grafu. Dva vrcholy jsou spojené hranou podle toho, jak spolu dvě příslušná políčka sousedí. Dále evidujeme obsazené vrcholy, které budou „obcházené“ algoritmem hledání. Graf reprezentujeme např. polem záznamů sousedů, kde pole může být i dvojrozměrné a záznam sousedů může být realizovaný čtyř-prvkovým polem (každý vrchol má maximálně čtyři sousedy).

10.11 Bludiště

Byla implementována úloha, která by se také mohla jinak nazývat „Cesty z Labyrintu“. Jak již jsme říkali, existují problémy v programování, pro které se řešící algoritmus nedá přímo stanovit. Pokud vůbec řešení existuje, dá se zjistit nanejvýš systematickým zkoušením. Tyto úlohy se proto většinou řeší metodou „pokus-omyl“. V tomto druhu řešení úloh se nejčastěji hovoří o metodě „backtracking“ (ústup). Tento postup poskytuje přístup do úplně nové úrovně problémů, kde hraje opět velkou roli rekurse.

Implementační část

Jak zobrazíme labyrint na obrazovce?

- Labyrint se skládá z cest, na které se dá vstoupit, a ze stěn. Pro jednoduchost by měly být všechny cesty a stěny navzájem pravoúhlé.
- Dvojdimenzionální pole k uložení půdorysu zpracujeme nejlépe jako **typizovanou konstantu**. Má to následující výhody:

Pro cesty, na které se smí vstoupit, používáme prázdný znak a pro stěny potom vybereme nějaký speciální znak.

Pro úlohu labyrintu na obrazovce používáme komponentu *TstingGrid*.

Princip algoritmu

Pro správné a systematické procházení všech cest bludiště jsme pro správný chod definovali některá pravidla:

- *Vycházejíce z aktuální pozice chceme pevně stanovit určité pořadí pro naše kroky: nejprve doprava, potom dolů, potom doleva a nakonec nahoru.*

- *Příští krok podle našeho pevně stanoveného pořadí může být proveden, jen když pole, na které vstupujeme není ani stěna ani východ.*
- *Už navštívená pole nesmí být navštívena znovu.*
- *Každé navštívené volné pole označíme a s označenými poli zacházíme, jako by to byly stěny.*
- *Když element cesty byl použit ve všech čtyřech směrech, je úplně jedno, jestli byl východ nalezen nebo ne, musí být toto značení vymazáno.*

Protože algoritmem jsou navštěvovány všechny elementy cesty, byly by při hledání východu označeny i cesty, které nevedou k cíli.

Metody nutné k manipulaci s kameny

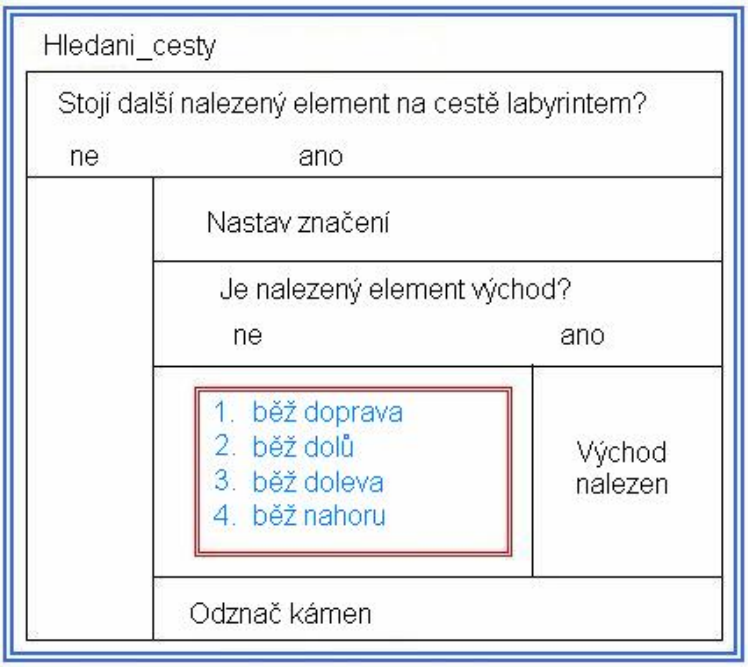
Vlastní manipulace s hracími kameny probíhá třemi metodami:

- *Nastav kámen*
- *Resetuj kámen*
- *Je kámen?*

Procedura *Nastav kámen* stanovuje umístění vybraného kamene na přesně označený bod hracího pole. Procedura *Resetuj kámen* vymaže vybraný kámen tím, že ho přepíše prázdným kamenem. Funkce *Je kámen?* zjišťuje, jestli se označený kámen nachází na označeném místě hracího pole.

Metoda, která „pohybuje kameny“

Teď přicházíme k vlastnímu „motoru“ celého programu, k proceduře hledání cesty. Nebudeme zde ale popisovat celý kód této metody, jen ukážeme podstatný strukturogram.



Strukturogram procedury pro hledání cesty v labyrintu

V každém kroku, kdy se kámen posune o další pozici kupředu, je testována procedura *Východ,* která při nalezení východu zastaví běh programu a čeká se, dokud se nestiskne tlačítko „pro další hledání“.

A co když hledání cest dospělo do slepé uličky?

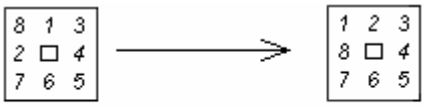
Žádná cesta nevede dopředu a žádná cesta nevede zase zpátky! Všechny pokusy na rekurzivním stupni dopadnou mylně. Předešlý strukturogram po vyzkoušení všech směrů odstraní značení. Tím je rekurzivní stupeň plně skončený. Teď budou vyvolány ještě nevyzkoušené směry na předposledním rekurzivním stupni. Protože všechny pokusy jsou chybné, bude značení opět odstraněno a zpracován předtím ležící rekurzivní stupeň.

Také všechny další pokusy na vzniklých rekurzivních stupních jsou chybné. Tím budou všechna značení až do konce hnědého orámování vzata zpět. Poprvé se ucelí backtracking.

10.12 Osmička

Hlavolam "*8*" má na desce o 3x3 políčkách osm průběžně očíslovaných kamenů, deváté políčko je volné. Přesouváním kamenů na volné políčko je možné postupně měnit uspořádání kamenů. Úkolem člověka (nebo "inteligentního" stroje) bude ze zadaného

výchozího (počátečního) uspořádání kamenů dosáhnout jejich postupným přesouváním zadaného cílového uspořádání, např.:



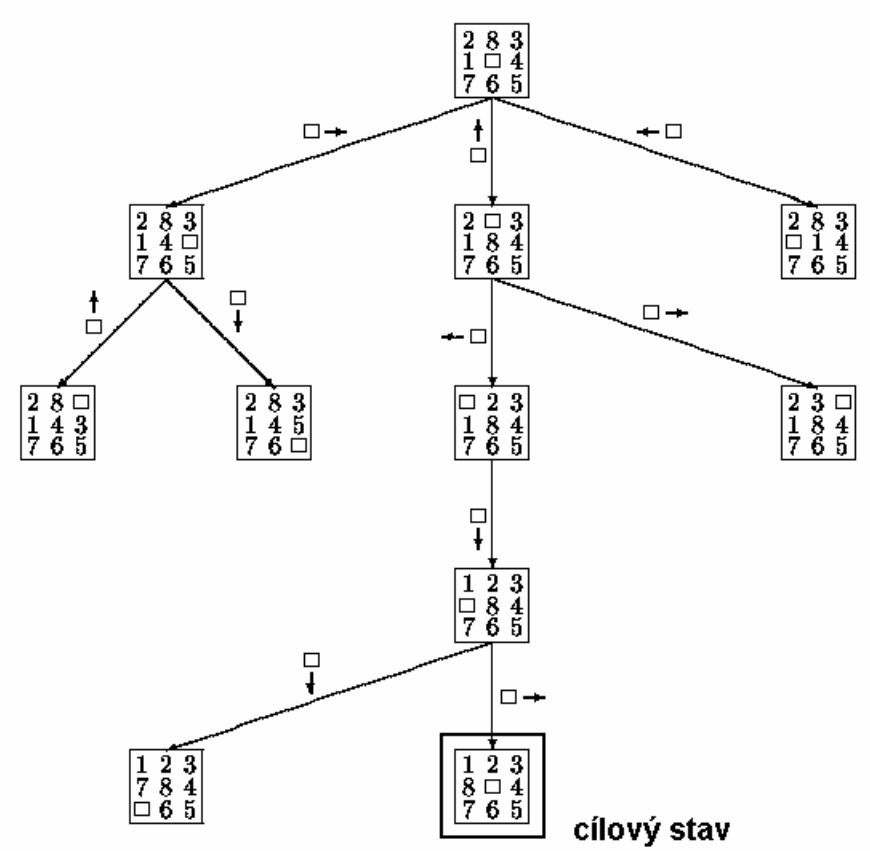
Obrázek 9.

Zřejmě jde o induktivní úlohu: výchozí (počáteční) uspořádání (stav) a cílové uspořádání jsou jednoprvkové množiny X, resp. Y, množina elementárních operátorů je dána pravidly hry, která bychom si mohli představit např. jako posuvy prázdného políčka (v obrázku prázdný čtverec) nahoru, dolů, doleva, doprava, přičemž prázdné políčko samozřejmě nesmí opustit hrací desku. Jednotlivá pravidla pro přesouvání kamenů mohou mít následující podobu (prázdné políčko nazvěme „blank“):

"blank" není v horním řádku - posuň "blank" nahoru,
"blank" není v dolním řádku - posuň "blank" dolů,
"blank" není v levém sloupci - posuň "blank" doleva
"blank" není v pravém sloupci - posuň "blank" doprava.

Pravidla hry spolu se zadáním výchozího a cílového uspořádání kamenů představují vše, co dokážeme z formulace úlohy získat. Je to informace značně neúplná, neboť na každý stav můžeme aplikovat vždy nejméně dvě pravidla a ze zadání úlohy nevyplývá, které z nich máme vybrat. Právě v tom spočívá nedeterminismus této úlohy. Nemáme-li žádnou informaci o tom, které z pravidel aplikovatelných na daný stav vybrat, musíme vybírat náhodně s rizikem, že náš výběr bude chybný. Pokud později chybu objevíme, musíme se vrátit až do místa vzniku chyby. Z toho důvodu si musíme místa, ve kterých přijímáme náhodná rozhodnutí, pamatovat a říkáme jim *uzly větvení*. Proces hledání řešení úlohy je pak *metodou pokusů a omylů.* Proces hledání řešení úlohy můžeme znázornit acyklickým grafem, který budeme dále nazývat *stromem řešení úlohy*. Uzly stromu jsou uzly větvení, hrany stromu představují jednotlivé alternativy postupu řešení (aplikace pravidel pro přesouvání kamenů), které jsme již vyzkoušeli.

Důsledkem nedeterminismu v řešení úloh je, že nalezení správného řešení úlohy může trvat velmi dlouho nebo proces hledání řešení dokonce neskončí vůbec. Proto se obvykle snažíme získat nějakou další doplňující informaci o úloze, na jejímž základě bychom mohli zformulovat kritérium umožňující vybírat jen takové hypotézy, které mají největší naději vyhovět daným množinám stavů a ty potom deduktivně prověřovat. Této doplňující informaci říkáme *heuristika*, protože pomáhá nalézt řešení. Na rozdíl od exaktních poznatků u heuristik přesně nevíme, za jakých předpokladů jsou pravdivé. Víme jen, že jsou pravdivé v běžných, typických situacích.



Během postupu řešení úlohy, tj. při aplikaci jednotlivých pravidel, hovoříme o přechodech úlohy z jednoho stavu do stavu dalšího, přičemž všechny stavy, které jsme prošli na cestě ze stavu výchozího do některého stavu cílového, nazýváme *vnitřními stavy* úlohy. Úloha je pak jednoznačně popsána množinou stavů, v nichž se může nacházet (a z nichž některé můžeme definovat jako stavy výchozí a jiné jako stavy cílové), a množinou pravidel pro přechody mezi stavy. Takový popis úlohy nazveme popisem *stavovým*, resp. *reprezentací úlohy ve stavovém prostoru*. Množinu stavů, v nichž se může úloha nacházet, nazveme *stavovým prostorem úlohy*.

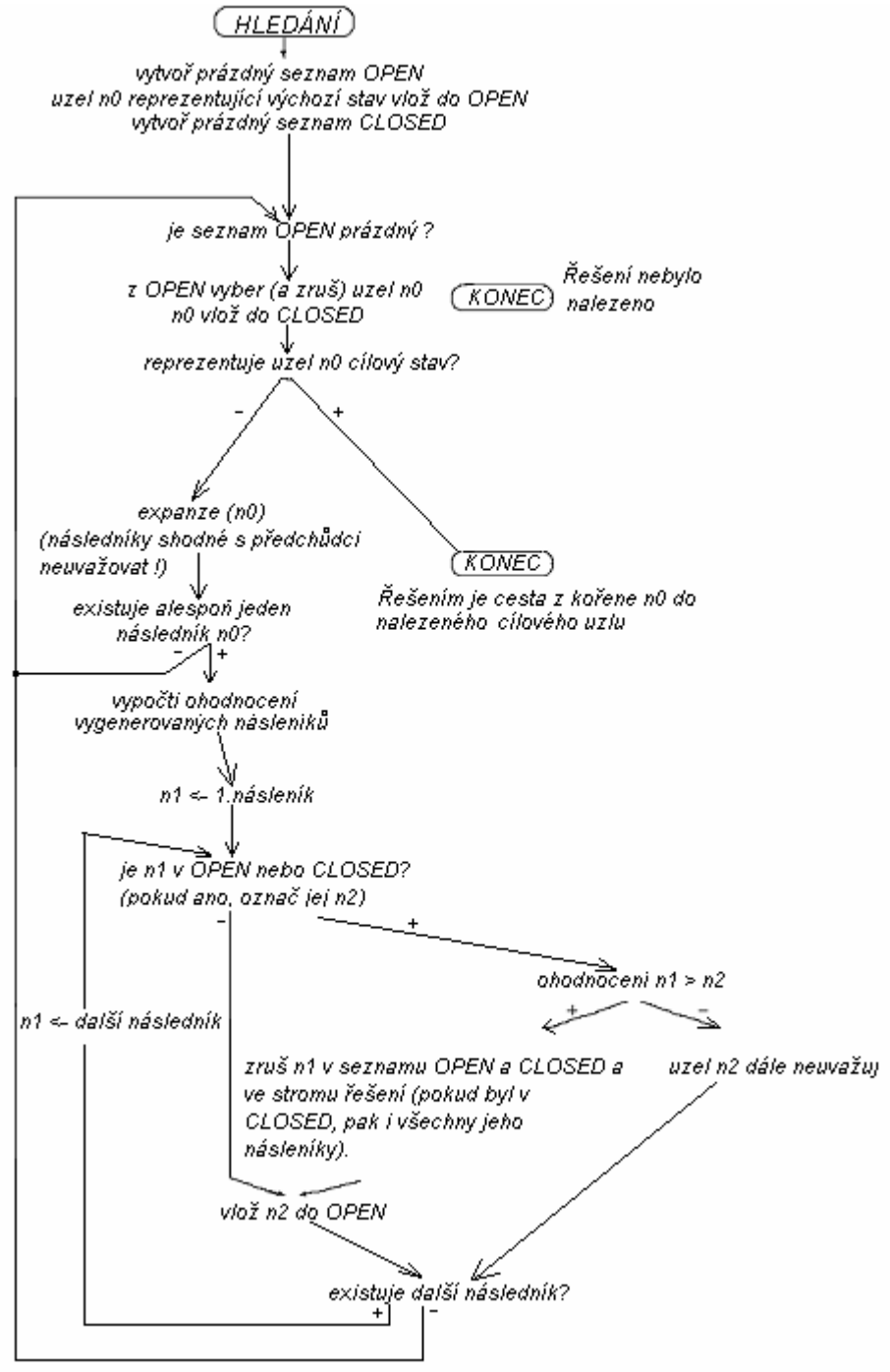
Implementační část

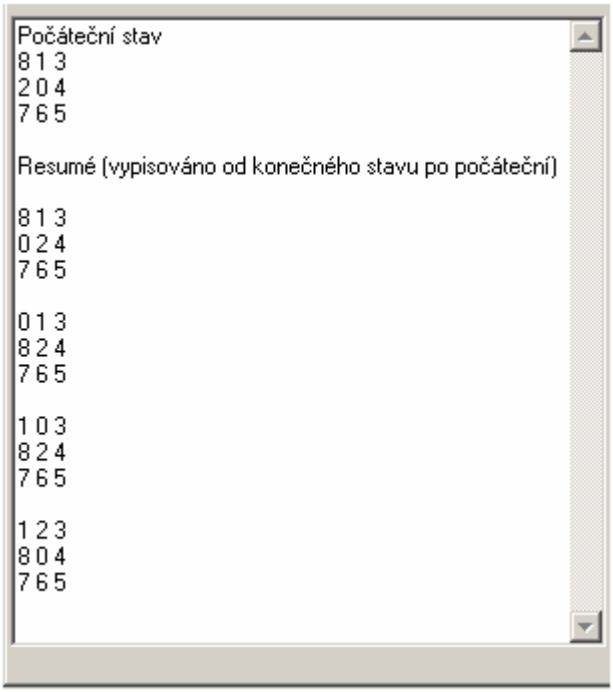
Základní *algoritmus hledání v grafu* si pro jednoduchost znázorníme vývojovým diagramem uvedeným níže.

Implementace algoritmu je založena na použití dvou seznamových struktur *OPEN* a *CLOSED*, přičemž v seznamu *OPEN* jsou uloženy uzly, které dosud nebyly expandovány[12] a v seznamu *CLOSED* se nacházejí uzly, které již expandovány byly

[12] Proces vygenerování všech možných bezprostředních následníků uzlu.

anebo se pro expanzi z nějakého důvodu nehodí. Dále budeme předpokládat, že máme k dispozici nějakou ohodnocující funkci, která pro každý obsah databáze(stav) vypočítá jeho kvantitativní ohodnocení.





Zde je znázorněn výstup aplikace dle obrázku 9. Je vidět, že k tomu, abychom se dostali z počáteční do koncové pozice, bylo třeba čtyř tahů.

11 Závěr

Diplomová práce řeší široké spektrum grafových algoritmů ať už základních nebo vybraných speciálních. Řeší tedy problémy týkající se hledání minimální kostry neorientovaného souvislého ohodnoceného grafu, prohledávání grafů, hledání nejkratší cesty a v neposlední řadě se zaměřuje na některé již praktičtěji pojaté úlohy. Mezi ně patří např. problém čtyř barev a úlohy založené na backtrackingu. V práci je zobrazena i funkce algoritmů nedeterministických, tedy algoritmů, které nemusí vždy dojít k nějakému řešení. Pakliže k řešení dojdou, pak za velice dlouhou dobu.

Ke všem řečeným algoritmům vznikla jejich příslušná implementace v moderním programovacím prostředí Delphi. Uživatel má tedy možnost vytvářet a modifikovat vlastnoručně nakreslené grafy a na těchto grafech řešit základní grafové úlohy a sledovat výsledky algoritmů okamžité nebo postupně zobrazované. Doprovodné úlohy kvůli jejich rozsahu obsahují určité ulehčení obsluhy uživatelů a jsou tedy již připravené pro samotné spuštění algoritmu.

Prostřednictvím těchto implementací si uživatel může ověřit nabyté teoretické znalosti.

Jelikož se jedná o jednotlivé vizualizace algoritmů, nemohly být oproštěny o některé grafické prvky a tudíž při měření časových náročností je třeba brát na toto omezení určitý ohled. U algoritmů se podařilo dosáhnout předepsaných časových náročností.

Realizované programové vybavení lze využít jako doplňující pomůcku při výuce základů diskrétní matematiky, operační analýzy a umělé inteligence na Technické Univerzitě v Liberci.

12 Literatura

[1] Demel J.: Grafy a jejich aplikace. Praha, ACADEMIA, 2002

[2] Kučera L.: Kombinatorické algoritmy. Praha, SNTL, 1989.

[3] Matoušek J., Nešetřil J.: Kapitoly z Diskrétní matematiky. Praha, Karolinum, 2002.

[4] Kolář J., Štěpánková O., Chytil M.: Logika, algebry a grafy. Praha, SNTL, 1989.

[5] Walter J. a kol.: Aplikace metod síťové analýzy v řízení a plánování. Praha, SNTL.

[6] Svoboda L., Voneš P.,Konšal T.,Mareš M.: 1001 tipů a triků pro Delphi. Computer Press, 2002.

[7] Wirth N.: Algoritmy a štruktúry údajov. Bratislava, ALFA, 1989.

[8] http://mathworld.wolfram.com/news/2003-08-06/magictours/